PRASE: INSTRUMENTATION SOFTWARE FOR THE INTEL iPSC HYPERCUBE

THESIS

Mark Albert Kahl
Captain, USAF

AFIT/GCS/ENG/88D-11

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/88D-11

PRASE:
INSTRUMENTATION SOFTWARE
FOR THE
INTEL iPSC HYPERCUBE

THESIS

Mark Albert Kahl
Captain, USAF

AFIT/GCS/ENG/88D-11

Approved for public release; distribution unlimited

AFIT/GCS/ENG/88D-11

PRASE:
INSTRUMENTATION SOFTWARE
FOR THE
INTEL iPSC HYPERCUBE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Systems)

Mark Albert Kahl, B.S.
Captain, USAF

December, 1988

Acknowledgments

There are several "Thank You's" I want to make, but one stands out in importance. My Lord and Savior Jesus Christ is loving and great beyond compare. He has taken me through this effort for which I am thankful. He deserves my highest praise and devotion. Thank you Jesus! In fact, I tried to find an applicable name for this thesis that would be a reminder of Him and at the same time fit the effort. I came close. He truly is wonderful.

I would also like to say thanks to my wife Jan. She has endured a lot here at AFIT while helping me get through it all. Her contributions to this thesis include editing suggestions, beautiful handwriting in an appendix, and tons of support. She has encouraged me, been my fan, and taken a lot of my jobs at home so that I would have more time to give to school. Thank you. I love you.

My children Tim, Lauri, Beth, and Melissa have all been great. They have been patient with the time required by school. Thank you all for your love. I am looking forward to spending lots more time with you all now that we are done.

My parents also deserve a "Thank You." Not only did they provide encouragement; they also provided my home computer which gave me much more time with my family. Thanks!

Next I would like to thank Dr. Hartrum. I think I ended up with one of the best advisors in all of AFIT. He was patient, calming when crises hit, and a great director of this effort. So much of this work reflects his ideas and thoughts. He deserves much credit. Thanks for always being willing to answer that "one last question." You were great!!!!

A special thanks also needs to go to Aiva Couch at Tufts University. He willingly supported us in providing Seecube and was excited about the results that

we got. He was encouraging and very willing to cooperate. I much appreciate your professionalism and cooperation. You contributed a lot to this effort.

Finally, I would like to thank some of the people here at AFIT who were willing to discuss this project with me. Some contributed by providing ideas that were incorporated in the thesis. Lots of input was provided by several individuals. First, my other two committee members; Cpt Davis and Capt Donlan. Thanks. Bruce Clay, and Rick Norris were very helpful. Their technical advice was excellent. Others include Capt Bailor, Cpt Shaw, Capt Hodges, Joe Glaeser, Capt Hippenmeyer, Cpt Lebano, and Capt Bridges. Thanks to you all.

Specifically, Capt Donlan, Cpt Davis, Capt Bailor, Capt Hodges, Bruce Clay, Rick Norris, Joe Glaeser, and Cpt Shaw were willing to 'interview' with me. Some of their inputs have been included in this thesis.

I know this has been a lengthy acknowledgements. However, I wanted to make sure that these individuals were named. An effort like this does not normally happen in a vacuum. Rather, it is the culmination of several peoples thoughts and ideas.

Mark Albert Kahl

Table of Contents

List of Figures

Abstract

PRASE (Parallel Resource Analysis Software Environment), developed at the Air Force Institute of Technology to support local users, consists of a set of subroutines and programs that aid a user in monitoring parallel processing software targeted for an Intel iPSC Hypercube. PRASE was in many ways patterned after Seecube, an effort by Alva Couch and others at Tufts University in Massachusetts. Like Seecube, instrumentation code must be embedded in a user's source code to facilitate data collection. After data has been collected, a translator may be used to translate PRASE data into Seecube format. Once translated, existing Seecube software can be utilized to produce several kinds of graphical displays on a Sun workstation.

Seecube, however, is not required to be able to use PRASE. PRASE allows a user to gather data on several processes per node and gives the user the capability to collect information on variables specific to his program. This allows application-specific instrumentation. Preprocessors for both C and Fortran automatically embed necessary subroutine calls using a user defined configuration file. The data collection subroutines are written in C and can be called by both C and Fortran. Data collected during program execution can be held in Hypercube memory and written to disk at the end of a run, or dumped periodically to disk during a run which may aid in debugging. The resulting data files can then be translated to Seecube format or used as input to other data analysis and display programs. The two preprocessors, as well as the translator were implemented in Ada.

PRASE:
INSTRUMENTATION SOFTWARE
FOR THE
INTEL iPSC HYPERCUBE

I. Overview

1.1 Background

Single processor computers are rapidly becoming a thing of the past. Their replacements are multiple processor systems, running in parallel, which may yield greater overall speed and efficiency. Mainframe computers sometimes use dual CPUs and almost always contain separate specialized processors to interface to input/output devices such as disk drives. Even some smaller home computers employ specialized chips (processors) to handle functions such as input/output or graphics.

There is great potential, in a parallel environment, to complete several times the amount of work of a single processor in the same amount of time. Just as a factory might hire more workers to increase production, computers can utilize multiple processors to better handle their workloads. In fact, when chip makers reach speed limitations, the only alternative for faster service may be to do more work in parallel.

The potential speedup offered by a parallel environment can be appealing to a user. A programmer, however, may not share the user's enthusiasm when faced with events happening at the same time and often in an unpredictable fashion. If parallel processors worked autonomously, the problem might not be so complex. Normally, though, they interact with each other, causing much greater complexity.

One way to attack the potential confusion is to use a parallel processing monitor/analyzer (PPM). A PPM should attempt to gather meaningful metrics (CPU utilization, message queue lengths, etc.) in a parallel processing environment and present the results in an orderly fashion. Without this type of tool, the programmer may face greater difficulty in understanding the parallel aspects of his software.

1.2 Problem Statement and Solution

A PPM, within the AFIT user community, has potential to enhance the programmer's ability to produce quality software in shorter periods of time. The goal of this thesis, then, was to provide a parallel processing monitor/analyzer that would collect data on an Intel iPSC Hypercube parallel processing machine, and graphically display the data collection results in a meaningful way on a SUN 3 workstation.

This thesis effort has resulted in the initial implementation of the Parallel Resource Analysis Software Environment or PRASE. Although the current implementation does not exist as a true Software Environment, this author forsees the utility of directing PRASE towards that end. Thus, the initial name includes some future goals for the system as a whole. The capabilities of PRASE are discussed throughout this thesis, but specifically in the System Analysis and Design chapter, Testing chapter, and the appendix containing the User's Manual. Compatibility with Seecube [2] is being utilized to meet the graphical display goal of this effort.

1.3 Summary of Literature/Local User Review

Three main sources were drawn upon to gather information pertaining to PPMs. First, a review of the literature brought out others' work and ideas relating to monitoring and parallel processing. Second, interviews with AFIT users were conducted to gather local opinions and thoughts. Finally, several ideas came from meetings with AFIT research faculty. Using results from these three sources, several questions were considered.

1. What performance measures are of interest?
2. What data should be monitored and how can it be collected?
3. How can the data be displayed so that it is easily understood?
4. How can a PPM be implemented to provide ease of use?

Systems such as Seecube [2] and Monit [9] have attempted to answer these questions. Seecube records message passing events on a Hypercube architecture to derive its performance measures of interest [2:1-2] while Monit gathers data pertaining to process events [9:165]. One part of the Seecube system, the *Resolver*, takes independently gathered data from each processor, and weaves it into a single file of data. This data can then be graphically displayed to a user on a Sun 3 workstation. A user may choose from many different display options. Monit also allows a user to manipulate gathered data using graphical displays [9]. Both systems are attempting to provide meaningful feedback to the programmers of parallel devices.

The book Measurement and Tuning of Computer Systems discusses several interesting ideas pertaining both to choice of measurements as well as to presentation of the results [6]. Even though these ideas are not given in a parallel context, many still apply to the PPM concept. The book lists measurements such as CPU utilization and memory utilization "from an earlier source" [6:14]. Other ideas dealing with the display of gathered data were also presented. Among the display ideas were Kiviat Graphs, which were discussed based on earlier articles [6:195-202].

Local users highlighted certain ideas as well as contributing new ones.[1] Some of their thoughts include time waiting for a message receive to complete, and correlation of message sends and receives to determine if messages were lost. Other thoughts were emphasized, such as finding message sizes, and reporting on queue lengths. Additionally, ease of use was considered as an important factor.

[1]See the Acknowledgements of this thesis for a list of the contributing individuals.

1.4 Scope

There were two items that were not a part of this thesis effort. First, no consideration was given to automated assessment of the collected data. This thesis was not meant to become a system that automatically evaluates the "goodness" of the data collected. Second, the PPM does not dynamically reconfigure the parallel environment. Monitored data is not used to tune the system dynamically. The goal was to provide a usable PPM to the AFIT user community.

1.5 Approach/Methodology

The initial phase began with the literature/user review. Many discussions with faculty as well as interviews with local users complemented what was found in the literature pertaining to PPMs. During the review, time was spent gaining experience working with both the Hypercubes and Sun 3 workstations. This time was used to become familiar with pertinent equipment and software. Throughout the project, prototyping efforts (small programs) were used to test new ideas.

The next two phases overlapped. Phase two was the design phase while phase three was the coding and testing phase. During phase two (and into phase three), several major decisions were made. Some of these decisions follow.

1. The Seecube software would be utilized in some capacity. (See the User's Manual for availability)
2. AFIT's own data collector[2] would be written (as opposed to updating the collector that came with Seecube or instrumenting the operating system).
3. Preprocessors would be provided to free the user from having to place most instrumentation code into programs by hand. Both C and Fortran preprocessors would be provided to support the two main Hypercube languages used here at AFIT.

[2]The term data collector has been picked up from the literature discussing Seecube[2:1].

4. Data collecting would not be in Seecube format. However, to maintain compatibility with Seecube, a translator would be provided to translate data collected with PRASE into the appropriate Seecube format to allow use of Seecube graphic display routines.

The third phase included writing and testing the PRASE software. Overlap with phase two occurred, in that several design decisions were made in tandem with coding and testing. As coding would progress, certain design considerations would become pertinent, at which time they were dealt with. Thus, several design issues were accomplished during coding and testing.

During the previous phase and into this phase, language choices were made. The data collector was written predominantly in C. Ada was chosen to implement the data translator, the Fortran preprocessor, and the C preprocessor.

Testing was incremental. As code modules were completed, they were tested. Thus, module testing was integrated throughout the coding phase. Finally, a full system test was conducted to check out the system as a whole. This test was done using a local user's code with the philosophy that actual data provides the best test case.[3]

1.6 Materials and Equipment

Two pieces of equipment are required to use PRASE. First, Intel iPSC/1 Hypercubes must be used to gather performance data in a parallel processing environment. This data can then be displayed using the Seecube display software which requires a color Sun ™ workstation.

[3]This type of testing philosophy was gained by working three years with a man I came to respect greatly - Ancel Peckham.

1.7 Summary

The goal of this thesis was to provide a parallel processing monitor to the AFIT user community. This was accomplished by providing new data collection routines and preprocessors, as well as a translation routine that supports the utilization of Seecube's graphical displays. AFIT Hypercube users now have another tool in their toolbox to aid in their parallel programming tasks.

II. Requirements Determination

2.1 Introduction

There are two questions, of those previously listed, that will be addressed here. The first is, "What performance measures are of interest?" Before a monitor can be effectively evaluated or designed, some sort of measurement guidelines must be set forth. With a standard in hand, design and implementation can take place.

Once data is collected, the second question becomes pertinent. It is, "How can the data be displayed so that it is easily understood?" Using the information collected in a literature review, coupled with the thoughts and ideas gathered in user interviews, the two questions of interest will be addressed.

2.2 Which Performance Measures?

What data or measurement statistics should a parallel processing monitor collect (or calculate from the collected data)? Whatever is collected, the ultimate goal of providing the user with helpful information should be kept in mind. Data should be selected for collection so that certain performance indicators can be quantified.

2.2.1 Performance Indexes [6:11-15] Several "Performance Indexes (sic)" were included by Ferrari, Serazzi, and Zeigner from an earlier source [6:14]. Some of these indices included *CPU utilization*, *channel utilization*, *multiprogramming level*, and *programs executed per hour*. Other indices suggested by the authors were *turnaround time* and *throughput* [6:11]. A short discussion of each of these indices follow.

CPU utilization was defined as the "Ratio between total CPU busy time and total system operating time" [6:14]. In a parallel processing environment, this ratio could be an especially effective indicator when used to compare one processor to another. The index could provide a first indication that only a few processors are

doing most of the work. Channel utilization was defined as the "Ratio between total channel busy time and total system operating time" [6:14]. This too, can provide an initial indication of an overworked area of the system.

Multiprogramming level is the "Mean number of simultaneously active (i.e., memory resident) programs" [6:14]. This could be used to compare the amount of processes running on different CPUs. High CPU utilization levels might be explained in the light shed by this index.

An important point should be made with the next index. *Programs* executed per hour [6:14] may be another measure from which a user could extract meaning. It would be very easy, however, to mistakenly say that the more programs per hour processed, the busier a CPU is. If a CPU's programs require only small amounts of resources, the processor with the most programs may indeed be the least worked. Great care must be taken in the interpretation of the gathered measures. Although interpretation is mainly a user function, a software monitor should not present data in a misleading way.

The definition for Turnaround time was given as "the time interval between the instant a program is submitted to a batch-processing system and the instant its execution ends" [6:11]. A use for this index might be the comparison of different program runs. A code modification may have increased, or decreased a program's "Turnaround time."

Finally, "Throughput" was "informally defined as the amount of work performed by a system in a given unit of time" [6:11]. This is another measure that can be used to compare processor performances within a parallel processing system.

2.2.2 Seecube Measurements [2] As explained by Couch and others, "Seecube is a development tool for parallel programming which uses postmortem records of local events from each processor to reconstruct the global state of the computer at any time during a computation" [2:1]. The *Data Collector* portion of the Seecube software tool obtains the raw event data.

The Seecube software intercepts send and receive message calls in a parallel environment where message passing is the means of communication between processors. An event is logged just prior to a send or receive call, as well as just after the call is complete [2:1-2]. Several data items are kept for an event such as, "the time of occurrence, a unique sequence number for the event, the identifying number of the processor sent to or received from, and the message type specified in the call" [2:2]. A receive event also logs the sender's sequence number to allow "Cross-referencing" of events [2:2].

Although message passing events are very important, they do not tell the whole story. Specifically, indexes such as throughput and memory utilization cannot be extracted from just capturing message events. However, whether these *other* indexes can be gathered in certain parallel environments is an important question. It may be that message passing events are the only easily gathered events within the system that Seecube runs on.

2.2.3 Monit Measurements [9:163-174] Early in its development, the Monit system dealt with gathering change of state data. Large amounts of data were produced. Kerola and Schwetman stated that "This tool proved to be a useful debugging aid, but it severely distorted the performance of the program because of the computing required to format the output lines" [9:165]. The developers moved on to a new design. This design used "a second form of data gathering, based on creating a file of time-stamped events" [9:165]. One goal in this change was to "minimize the resources consumed by this logging activity" [9:165].

The designers were "initially" interested with extracting data that would help them find out "where PPL processes were spending their time" [9:165]. (PPL is the name of a "parallel programming language" [9:164]) They "defined events corresponding to assigning and unassigning processes to worker tasks and to processes entering and leaving queues associated with the synchronization and interprocess communication facilities of PPL" [9:165]. Unlike Seecube, Monit appears to center more on gathering data pertaining to processes rather than messages.

2.2.4 Local User Input Discussions were held with local users resulting in a list of several items of interest.[1] Some of the self-explanatory items were message size, amounts of messages (sent, received, handled), number of messages waiting at a given time, queue lengths (average and maximum), waiting times, memory utilization, CPU idle time, and CPU utilization.

Several other ideas were presented. Some of these are enumerated below.

1. Check to see if the message arrived. How many messages were sent that didn't arrive at their destinations? [8]
2. Correlate message types. This could include statistics for each message type (lengths, numbers, etc.). [11]
3. Follow a message. Show the path of a message as it flows through a parallel system. [11]
4. Allow a user marker[2] facility. A user could *mark* a point in a monitor trace to identify what might be happening at the time of interest. This idea can be extended to allow a user to write values or text to a monitor trace for later analysis. [5]

[1] See the Acknowledgements of this thesis for a list of the contributing individuals.

[2] The term marker or mark was also found in the Seecube literature pertaining to something similar to the idea presented here.

5. Provide waiting time statistics for message passing utilities. For example, how long did a program wait from the time a receive and wait was issued to the time the receive completed? [5]
6. Provide load balancing information. [1]
7. Provide speedup information. [1]
8. Determine the communication compute ratio. [1]

2.3 Communicating Results

Effectively communicating results encompasses more than merely choosing the best graphs, charts, or tabulation methods to display data collection results [6:175]. Display techniques may also be important in aiding a user's understanding. Ferrari and others state that "the most important problem is not so much that of determining the number and type of the graphs to be produced, but that of how to group the various quantities in the tables and diagrams" [6:175]. Ways of displaying information will be discussed in the next four sections.

2.3.1 Gantt Charts and Kiviat Graphs [6:192-202] In the literature, examples of Gantt charts, Kiviat graphs, line graphs, histograms, bar graphs, and data tables were found [6:174-203]. Gantt charts and Kiviat graphs will be discussed in detail. Figures 2.1 and 2.2 have been copied from the book *Measurement and Tuning of Computer Systems* [6].

"Utilization Profiles," a form of Gantt chart, "represent simultaneously in a single diagram the most significant variables of a system's activity" [6:192]. As displayed in Figure 2.1, these profiles provide a means by which events that overlap can be shown. As discussed by Ferrari and others, Figure 2.1 shows, "a system ... that had a CPU activity overlapped with that of at least one channel 22 percent of the total time" [6:192].

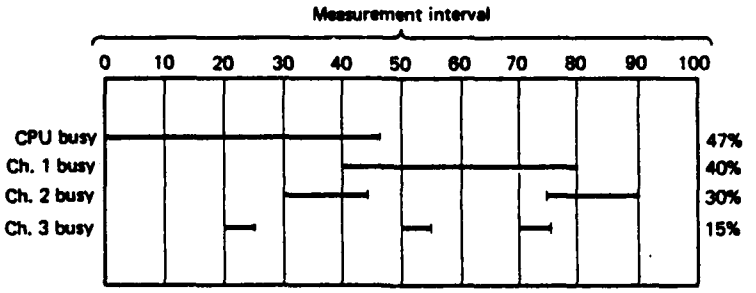


Figure 2.1. Utilization profile (Gantt chart). [6:192]

More generally even then "Utilization Profiles," Gantt charts or some form of time line could be used very effectively to demonstrate time ordering of events.

Ferrari, Serazzi, and Zeigner stated the following about Kiviat graphs:

> Kiviat suggested that several variables could be reported on semiaxes irradiating from a point, called a pole, and that the points corresponding to their values (according to predetermined scales) could be connected by straight-line segments, thereby obtaining a polygon, called a Kiviat graph. A Kiviat graph can be drawn when a circle, whose center is the pole, and three or more semiaxes from the pole are given. The intersection of the circle with a semiaxis corresponds to the maximum value of the variable being displayed on that semiaxis. The number of semiaxes is arbitrary and generally selected according to completeness requirements for the data represented; ... [6:195]

Figure 2.2 shows both a Kiviat graph and a corresponding Gantt chart for the same data.

Both Kiviat graphs and Gantt charts provide a designer graphical means of representing data pertaining to a parallel processing system. Either or both of these may be possible display options for a PPM.

2.3.2 Seecube Data Presentation The *Sequencer* portion of the Seecube software is responsible for preparing and presenting the collected data to the user [2:2-5].

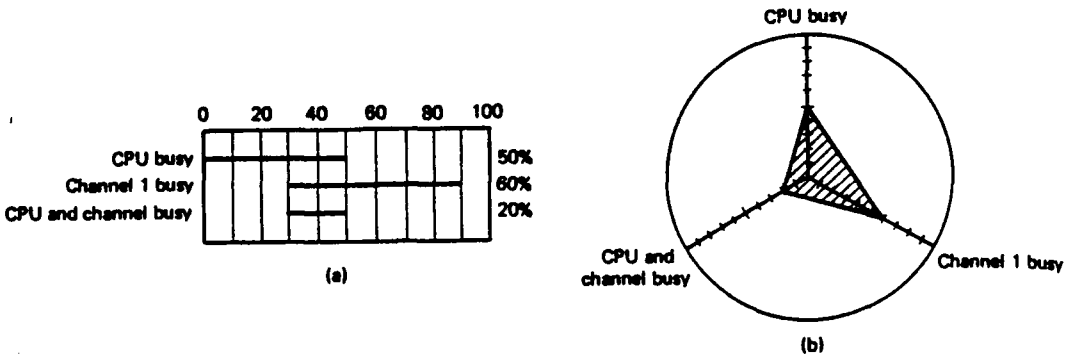


Figure 2.2. Example of a utilization profile and a Kiviat graph for the same workload conditions. [6:196]

This is done by dealing with states. The author's working definition of a state was, "the value of any performance parameter of the hypercube which is constant between events and can be inferred using only the event trace" [2:2]. The hypercube is the parallel processing architecture on which the Seecube software collects data. The authors said that they "...chose to reconstruct states rather than directly displaying events because of the problems of graphically representing instantaneous events on dynamic displays." They went on to say that, "As each state by definition always has a duration, states are ideal for use on time-varying displays" [2:3].

The *Sequencer* allows the user to choose the "global time to display" [2:3]. It also allows the user to move forward and backward in time. Couch and others say that "The real power of the Sequencer, though, lies in its ability to manage and juxtapose several different displays of the same states on one graphics screen" [2:3]. Another facet of the *Sequencer* is the way in which states are displayed. The authors said that "To display these states, we decided upon an equally simple convention: each parameter we compute is represented as a shape on a graphics screen, and the

value or state of that parameter is represented as the color of that shape" [2:3]. They also explained what colors could represent. They said that "The color of a processor symbol can correspond to the processor's internal state (sending, receiving, computing, or idle), to the number of messages which are currently in process of being routed through that processor on the way to their destinations, or to the total size of all messages being routed" [2:4]. The color or highlighting idea seems useful. It could be used to draw a user's attention to an area that is much different than the average.

Seecube is able to present several different kinds of displays pertaining to the data collected. We will discuss three of these displays briefly. First, the Seecube Sequencer provides a "Cube Display" which represents all the nodes within the Hypercube and their associated communications channels [3:14-22]. By manipulating certain parameters on this display panel, one can examine the data in different ways. This display attempts to show the total message activity within the system by considering a message to be active on all communication channels from originator to destination. This means that if two intermediate nodes and their associated channels are used to transmit a message, this display will present the activity. [3] An example of this display can be seen in the Figure 2.3.

Another seemingly useful display is the "Clump" display. Whereas the cube display attempts to show total message activity, the clump display aims at displaying originator to destination activity [3:22]. A user is presented with a grid that represents point to point communications. Each individual node can be represented, or nodes can be "clumped" together to show message traffic from one group of nodes to another [3:22]. An example of this type of display is provided in Figure 2.4.

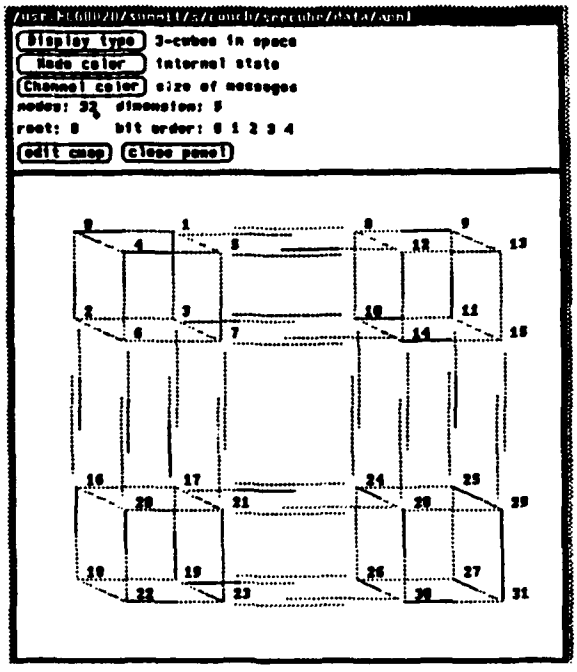


Figure 2.3. The "**3-cubes in Space**" Representation of a 5-d Hypercube Using the *Cube* Display. [3:15]

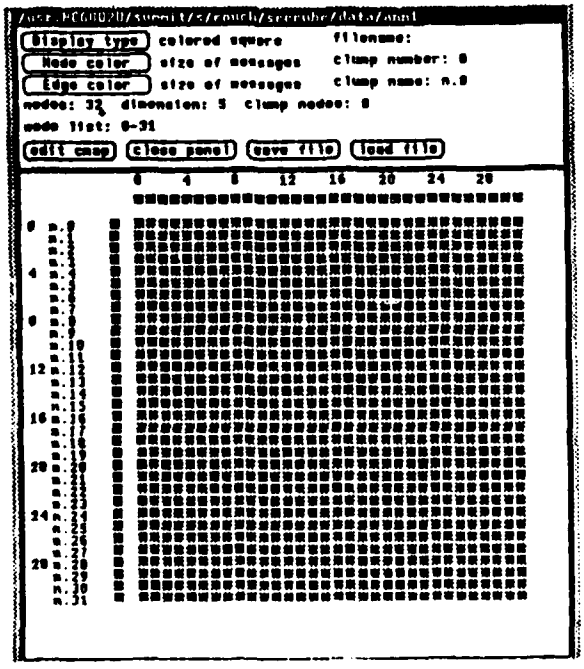


Figure 2.4. The "**Colored Square**" Representation of a 32 Node Complete Graph Using the *Clump* Display. [3:23]

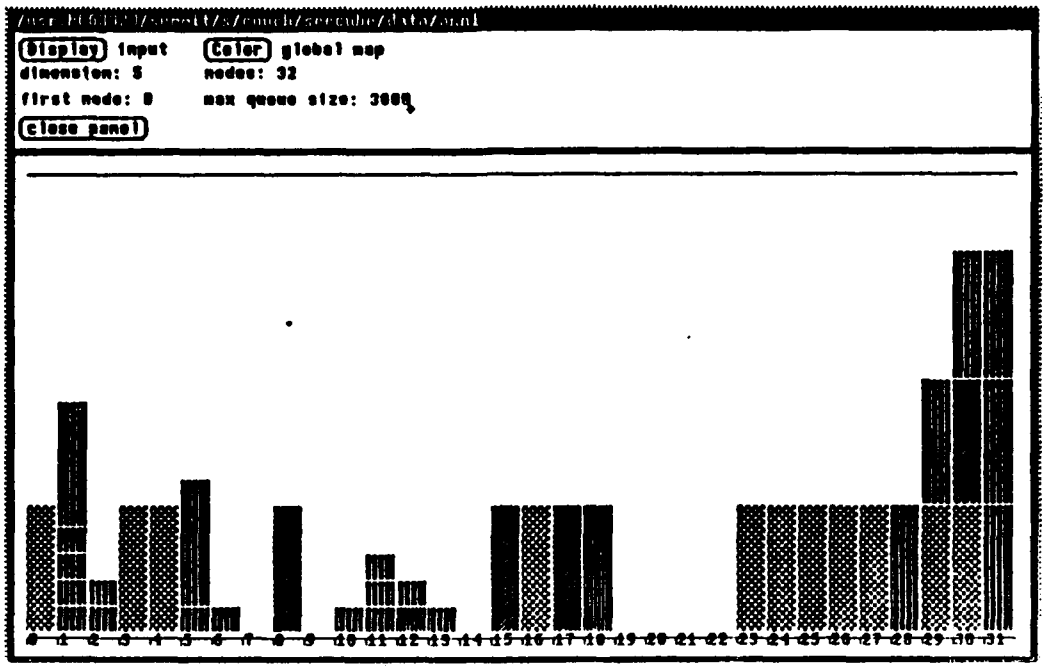


Figure 2.5. The *Queue* Display of Input Activity. [3:32]

A third useful display is the queue display. Here, queues of messages at each node can be represented. A user can watch how queue sizes grow both in numbers of messages and size of messages. This display might highlight a backlog problem that was previously undetected [3:30-33]. Figure 2.5 is an example of this type of display.

Seecube provides a tailorable environment which allows the user to pick and choose items of interest. More than one copy of the displays discussed can be represented at once [3:14]. For example, a user might wish to view two queue displays; one showing output queues and one displaying input queues [3].

2.3.3 Monit Data Presentation According to Kerola and Schwetman, a user of Monit "can specify the interval of interest by the zoom parameter and the display resolution by the step size parameter." The user "can also change the widths of bar charts by modifying the display accuracy parameter" [9:166]. Monit mainly displays results using "a columnar bar chart, consisting of many horizontal lines"

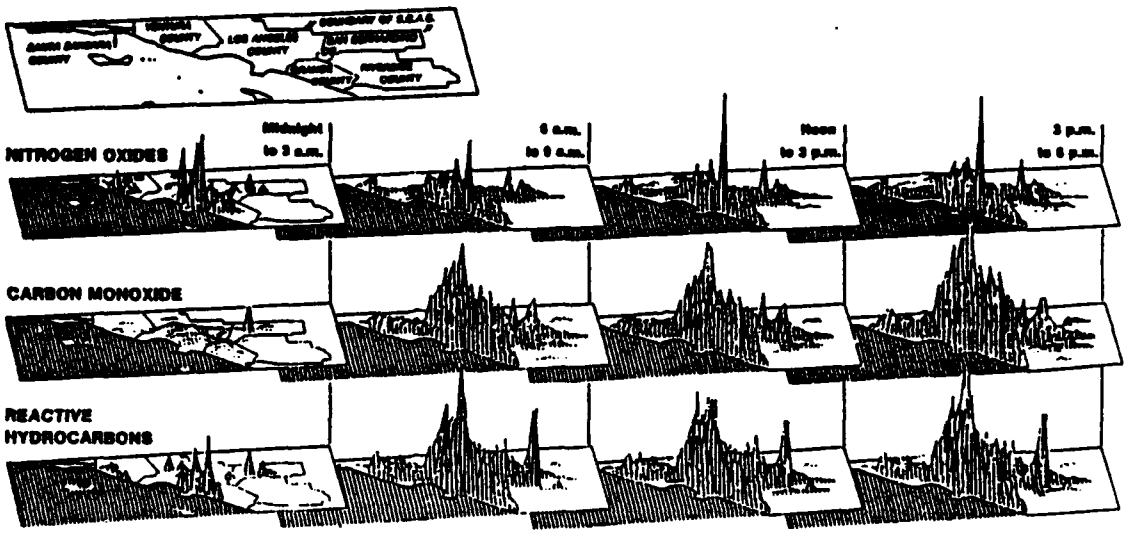


Figure 2.6. Air Pollutants Graphs (no title found in source). [13:42]

[9:166]. In reference to the bar charts, the authors say that "The length of each line in a column corresponds to the average value of the displayed item over the time interval determined by the current step size." Monit also provides a table of "conventional statistical data (mean queue lengths, response times, etc.)" [9:166]. The authors state that when "too many similar objects" are needed to be displayed, Monit can group them to "display only the maximum, minimum, or the average of the associated values in the group" [9:166].

2.3.4 Three-Dimensional Line Plots Early in this thesis effort, the idea of a 3-D line plot was considered. The plots would be used in a time-varying mode. One axis could represent time in selected increments. A second axis would represent the process where the data was gathered. The third (and final) axis would represent the measurement of interest. With this type of graph, areas of concern or at least areas that are significantly different should become obvious. Figures 2.6 and 2.7 were subsequently found in the literature.

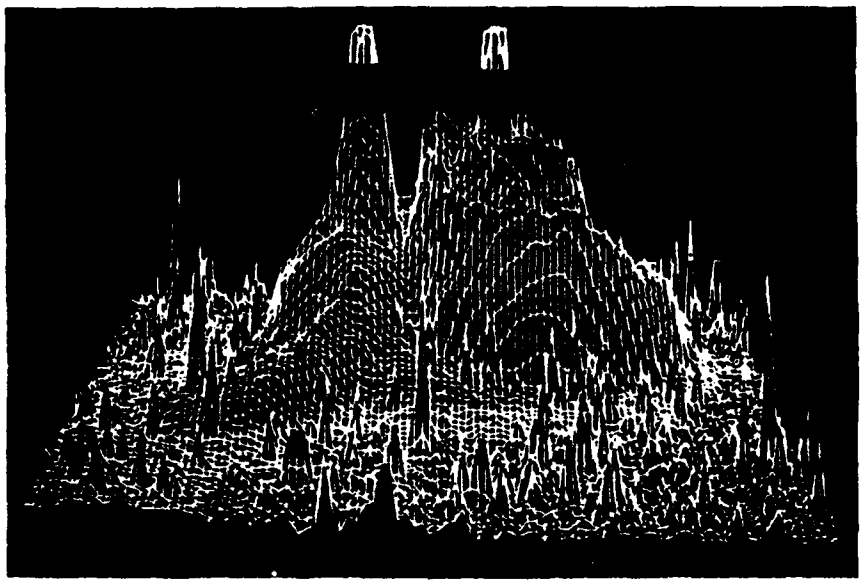


Figure 2.7. A color-coded plot with 16 million density points of relative brightness observed for the Whirlpool Nebula reveals two distinct galaxies. Courtesy Los Alamos National Laboratory. [7:10]

Figure 2.6 was originally taken from the *Los Angeles Times* July 22, 1979 issue. The graph was apparently based on work done by Gregory J. McRae, California Institute of Technology.

Figure 2.7 was printed in the book referenced courtesy of Los Alamos National Laboratory. Although not evident from this black and white rendition, Figure 2.7 is color coded to enhance the change in height on the graph.

2.4 Summary

This review has concentrated on two issues: first, what to measure; and second, how to present the data once measured. Seecube and Monit provide examples pertaining to both of these areas.

III. System Analysis and Design

3.1 Introduction

There were several directions this project could have taken. The next two sections will discuss these alternatives and the choices made. Section 3.4 will address the overall system design and certain major design considerations, including PRASE's relationship to Seecube. The fifth section of this chapter will explain more detailed design issues pertaining to some parts of the system. Finally, Section 3.6 will briefly discuss some of the ideas from Chapter Two that have been implemented.

3.2 Alternatives

As mentioned previously, several courses of action presented viable alternatives. Prior to addressing final decisions, a short discussion of these alternatives is in order. Four possible directions will be presented.

3.2.1 Operating System Instrumentation AFIT owns a source license for the node operating system running on the iPSC/1 Hypercube. Thus, hooks and monitoring subroutines could have been placed directly into the node operating system. On the positive side, this would have allowed near total transparency for the user as well as the ability to measure some parameters not otherwise available. On the negative side, once the operating system was upgraded, any modifications would have become obsolete. Another problem concerns the difficulty of trying to change possibly very complex code. Finally, portability would be limited to other sites owning a source license.

3.2.2 Seecube Upgrade The Seecube code, along with documentation, was obtained from Tufts University in Massachusetts.[1] Since the source code was available,

[1]See the User's Manual for details pertaining to obtaining a copy of Seecube.

modifications could have been made to include enhanced capabilities. A positive aspect of this option was that Seecube was an existing system that already provided some sophisticated capabilities. The negative aspects began with the fact that the Data Collector (subroutines that facilitate data gathering) had not been updated for the current version of the Intel operating system. This would have meant trying to understand and modify another's code. Also, any copyrights for Seecube would have prevented the work from being solely under AFIT's jurisdiction.

3.2.3 Autonomous New Effort A third alternative was to go completely away from anything current, and begin again. A positive aspect, at least for this author, meant not having to understand and modify another's code. Also, the project would be AFIT's, providing any freedom coupled with single ownership. The negative aspect to this was that no previous coding efforts would be utilized.

3.2.4 Some New, Some Old This alternative dealt with starting a new effort but maintaining compatibility with Seecube. The positive side of this alternative includes sole ownership, as well as reusing some of a previous effort. This option also provides the opportunity to start afresh in the design and coding effort. On the negative side, some software would be redone. Also, to maintain compatibility, either the same data formats would have to be used or a translator would have to be provided.

3.3 Choices

At some point a decision had to be made. Each of the listed alternatives had some merit, with really none of the alternatives being unacceptable. Thus, the decision was made to start a new effort but maintain compatibility with Seecube. By going this way, there was flexibility in the way the system could be implemented. This also relieved the burden of understanding and updating another's code. The Government would have sole ownership and any flexibility desired in dealing with

the final product. New capabilities and ways of dealing with certain issues could be handled without the burden of molding those capabilities into someone else's design.

Certainly, the Seecube software could have been modified to support any new data requirements. The display software could also have been enhanced to support this new data. However, the freedom to start afresh (as opposed to changing/updating another's code), was a major factor, if not the major factor in this decision.

A new Data Collector would be written and a translator provided to translate the data into Seecube format. Thus, data collected could be different than what Seecube collected as long as certain items were provided to support the translation process. The main purpose for maintaining compatibility was to utilize the display software provided by Seecube. By collecting new and possibly different data, new and autonomous display software could also be developed.

Going with Seecube meant staying in the realm of event detection as opposed to sampling. An event detection monitor collects data upon a specific occurrence of some event. [12:22] The alternative was to sample data as opposed to collect data on specific events. "A sampling monitor is similar to an event driven monitor but it is activated by an interval timer, collects data about the system for a set period of time and then is deactivated by a timer." [12:22]

Preprocessors would be provided to relieve the user of much of the burden of instrumenting his own code. These programs (one for Fortran and one for C) would accept, as input, code that compiles (no syntax errors) on the Hypercube. The output would be code with instrumentation software added. This would decouple the user from some of the details needed to use the new system.

3.4 System Design

It should be noted that Seecube provided an excellent springboard for this project. Several ideas came directly from the Seecube effort. For example, the

separate subroutines to collect data are very much like those found in Seecube. The data collected also greatly resembles the data collected by Seecube. However, they are not the same. The PRASE data formats, synchronization methods, and certain other items are different.

The overall design of the system incorporates the use of Seecube in concert with the new software. Figure 3.1 shows how the PRASE code fits together as well as how the connection is made to the Seecube software. Prior to listing major design issues, a review of how the system is used may help to put the design into perspective.

3.4.1 Using PRASE - An Overview The user must first take compilable code and run it through a preprocessor. Most language constructs[2] in Ryan-McFarland Fortran are supported by PRASE. This allows the user to be mostly freed from the burden of understanding what must be added to his code to use PRASE. The C language is also supported by a separate preprocessor. One 'special' comment must be manually inserted in a C program prior to preprocessing.

A configuration file is used by both preprocessors as well as the translator to set the context in which the programs will run. Such information as which nodes are running, what process *ids* are being used, and what types of information to gather is included in this file. The user must prepare this information using a standard text editor to use the PRASE system.

Data may be gathered pertaining to more than one process per node. However, if the end result is to use the Seecube displays, only one process per node is allowed. This is because the available version of Seecube limits the user to one process per node.

Once the user's program has been converted, it is ready to compile. An *include* file for PRASE must be updated, and a personal copy of the subroutines must be

[2]Not all language constructs are supported. The user should refer to the User's Manual for details.

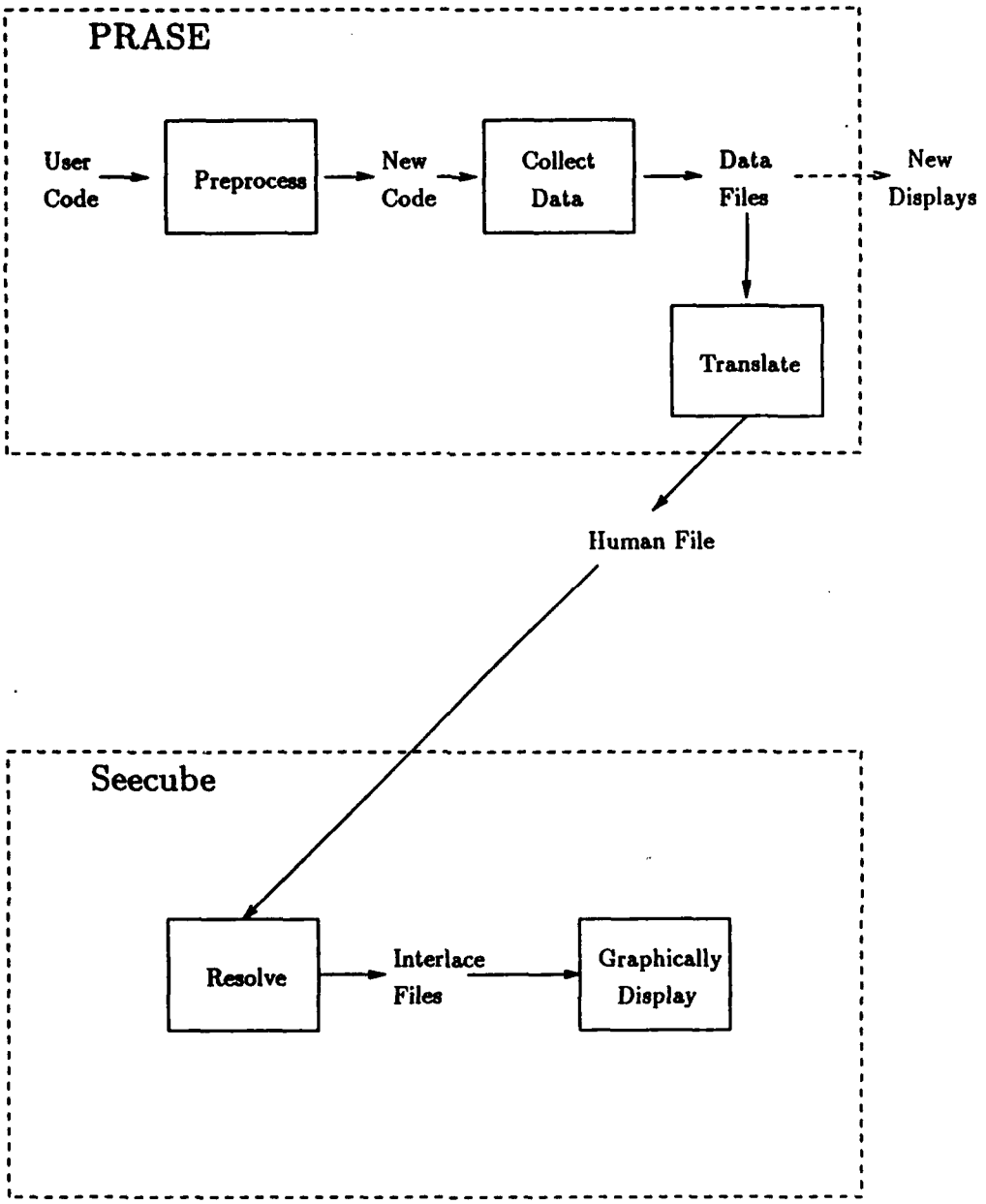


Figure 3.1. PRASE and its Connection with Seecube

moved into the user's local directory. The user can then tailor his own copy of PRASE to fit his requirements. The user should then update his makefile, and compile the new code in preparation for running on the cube nodes. It should be noted at this point that only node programs are instrumented. Message sends and receives in reference to the host are recorded; however, the corresponding activity within the host program is not monitored because data collection facilities for host programs do not currently exist.

Once the routine is ready to load, cube operation can begin. A collection program must be run in the background on the host to accept the data as it is dumped from each node process. Once the data is accepted and dumped to file, it is then ready for translation and analysis. See Figure 3.2.

The data collected can be manipulated directly by an analysis program or translated for use in the Seecube environment. As can be seen from Figure 3.1, the data is translated into a format that is accepted by the Seecube Resolver. The Seecube Sequencer can then be used to graphically review the collected data.

3.4.2 Major Design Issues Various issues had to be dealt with in the design of the PRASE system. The following list embodies several of these issues including a short discussion of each.

1. Collection Routines Implementation - The data collection routines were implemented in C. These routines gather data pertaining to activity within the Hypercube. Normally, a time prior to an action is gathered, a time after the action is gathered, and then the pertinent data within the call is collected into one record and stored in memory until a dump is required.

   Instead of implementing another set of instrumentation subroutines in Fortran, the C routines were called directly from Fortran. This has a potentially serious implication. No longer are calls being made to the Intel iPSC Fortran node routines. Instead, the Intel iPSC C routines are being called indirectly. Calling

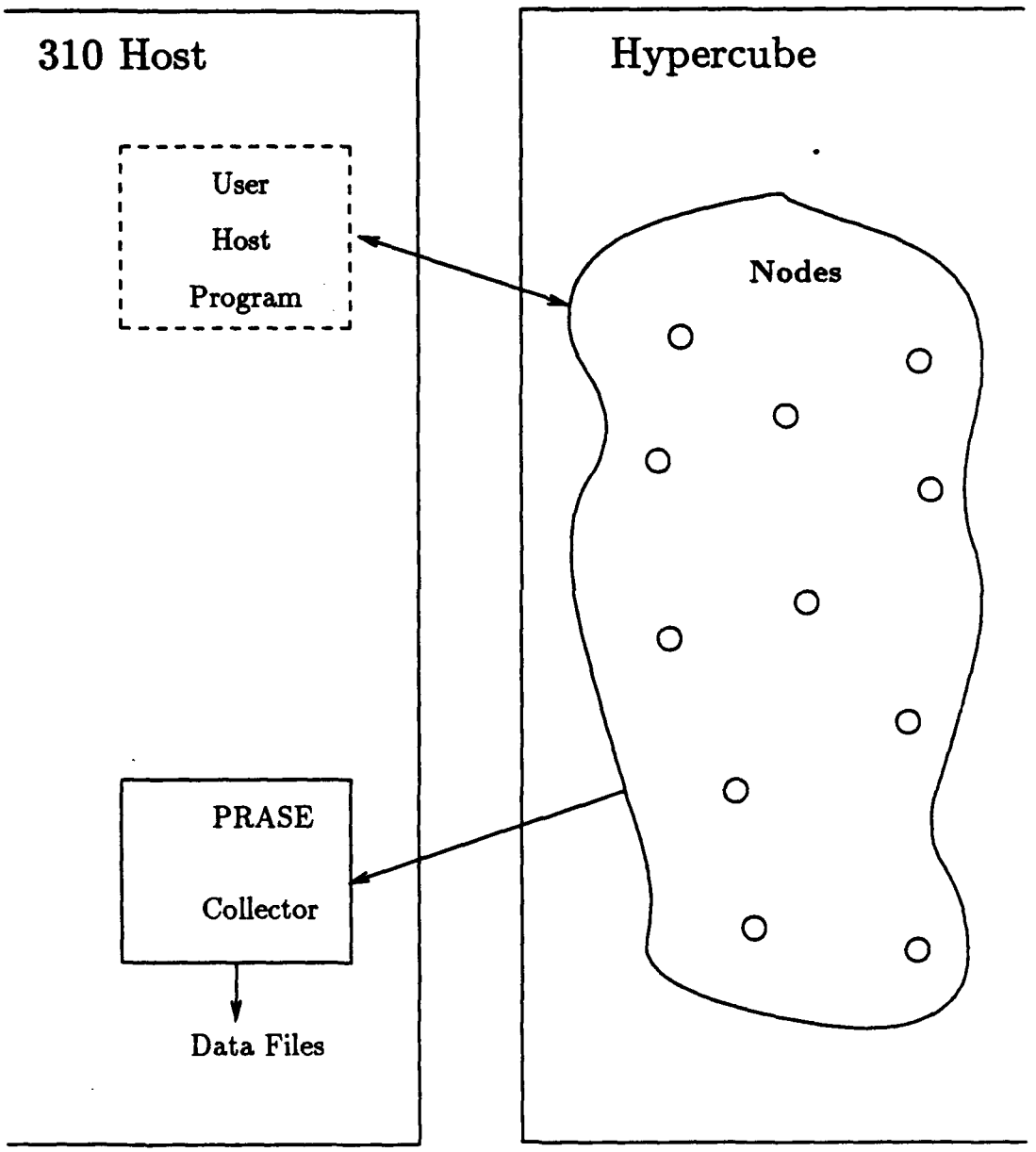


Figure 3.2. Data Collection using a Background Process

different routines may mean that performance is different when instrumenting as opposed to when no instrumentation is included. However, whenever additional code is used to gather data, performance is altered. Because of the time saved by not having to implement a set of instrumentation routines in Fortran, the risk was considered acceptable.

2. Configuration File - This approach was chosen as the method by which the user could specify certain information to the PRASE system. Information such as which nodes are active, what process id numbers are running on certain nodes, what items to instrument, and what file names to preprocess are included in the configuration file. This file is used by both the preprocessors and the translator.

3. Data Collection - Somehow, data collected on a node must be written to file on the host. This data could be written directly to disk from the nodes using prototype code provided by Intel. Another option might be to force the user to run his own host program that calls a collection subroutine provided by PRASE. A third option, however, was chosen.

   A collection program, provided with PRASE, is started in the background on the host. This program then waits for all messages destined for the host with a predetermined process id number. This process id must be unique (not used by the user). The dump routine is called by the node processes when certain thresholds are reached. This routine sends the data to the special host process *id* so that the collection routine can gather and store the data.

   The user is allowed to set two thresholds pertinent to data collection. The first is the number of records to keep for this run. The user specifies in a separate file (not the configuration file but rather an include file for the collection routines to use) how many records of data are allowed for this run. The second threshold set by the user is how often dumping is accomplished. A user can specify this threshold as 1, causing the collection routines to dump after every event. This

type of collection might be useful for debugging but is very expensive in terms of message passing overhead. By being able to set how often to dump, a user can save few records in memory and dump often allowing him to save as much data as the disk drive will hold. He is not limited by the onboard main memory.

If the user sets this second threshold greater than or equal to the maximum number of records, then dumping will not occur until the completion of the run. For time critical applications, this is the preferred method.

If an application tries to collect more data records than the user designated, PRASE will automatically cut off collection without affecting data records already collected. However, if a user asks to collect more data records in memory than the memory space available, the Hypercube will provide an error message in the *log* file when loading to the nodes is attempted.

4. Data Storage Format - Data was stored both in memory and on disk in binary format. This was done to save space. Because of this decision, a small program was needed to view the data in an ASCII format.

   The decision to store data in a binary format caused problems later in the effort. The hypercube stores binary data in a certain order (low byte first). The Sun workstation, where translation would occur, stores binary data high byte first. Therefore, extra manipulation of the data was required because of the storage format chosen.

5. Ease of Use - Preprocessors were included in the system to aid the user with the task of changing his code. This was done to make the system easier to use. This frees the user from having to understand all the details of the PRASE system.

6. Global Variables and Routine Names - In order to avoid names that might interfere with a user's application, reserved words begin with some form of the word PRASE (either upper or lower case). Thus, as long as a user does

not implement variables or other names with these characters, the applications program should be free from any conflict with the collection software.

7. Send and Receive Correlation - Seecube required a user's send and receive message buffers to contain additional bytes at the beginning for Seecube use. These bytes were used to send cross-referencing information between nodes. These bytes are not required within PRASE.

   PRASE implements a message count structure within each process. There are separate structures for send and receive messages. Basically, a sequential count is kept of each unique triplet of message type, corresponding node, and corresponding *pid*. This count is included in the data records as they are saved. Since the Hypercube keeps specific message types sent to or received from a specific node and *pid* in order, the previously mentioned triplet coupled with the sequential count, uniquely identifies a message on a single node.

   To restate this idea, six pieces of information uniquely identify a message. They are message type, local node, local process *id*, corresponding node, corresponding process *id* (*pid*), and the sequence in which the message was sent/received. The sequence counts are in reference to the beginning of a run. Even if tracing is forestalled until some time after the start of the run, the counts are kept, starting from the first message sent/received (whether the message event was recorded or not). Any count has some limit depending on the word size being used to store that variable. To remove this restriction from the count, a special wrap data record was implemented. At some predetermined point, PRASE recognizes that the count has grown big enough. At this point, a wrap record is written to the data trace and the count reset.

   Using this type of implementation, the number of messages can theoretically go on to infinity (if the maximum number of records parameter is ignored for the moment). The translator, however, does not count wraps forever but does

allow a large amount of wraps prior to overflow. For a more specific description of the overall count process and correlation, see Appendix G.

A basic assumption is made here. It is that the iPSC Hypercube will not lose or intermix messages of the same type. In defense of Seecube, its method is much more general and more robust in the event of errors. It was decided, however, that for PRASE that risk was acceptable enough to relieve the need for adding the extra bytes at the beginning of each message.

8. Time Synchronization between Nodes - Synchronization is performed in a simple way. At preprocessor time, the lowest node number running processes is identified. The first process *id* listed in the configuration file for this node becomes the "controlling process." This process takes charge of synchronization.

   The controlling process (or CP) goes through the list of other processes that are running for this current application, and communicates with them one by one. All other node processes running will perform a receive and wait message service and wait for its turn to synchronize. To illustrate the process, node 1 process *id* 1 will be used and will henceforth be referred to as $N_{1,1}$. CP contacts $N_{1,1}$ to let it know that it is its turn to synchronize. $N_{1,1}$ then sends a message off to CP and waits for a reply, meanwhile saving the time (T_{SEND_1}) that the message was sent. CP accepts $N_{1,1}$'s message, fills it in with a time of receipt ($T_{RECEIPT_1}$), and immediately sends it back to node 1. As soon as node 1 receives the message, it records a second time (T_{SEND_2}) and begins to calculate.

   The total message transit time is determined by subtracting T_{SEND_1} from T_{SEND_2}. A one way transmission time is then calculated as one half of the total transmission time. Thus $N_{1,1}$ can now figure out an assumed time that its synchronization message arrived at CP. This time will be called T_{HALF}. Since CP sent $N_{1,1}$ the time $T_{RECEIPT_1}$, $N_{1,1}$ can now determine a difference between its clock and CP's clock. Once done, we have synchronized this node

with CP. This time difference is then added to the base time (CP's start time) on CP.

The result of all this is a single time that is subsequently subtracted from all other times gotten directly from the node clock. When a time is read from the node clock, and the subtraction time is subtracted, the result is a time that tells approximately how many milliseconds have elapsed from the start of the run.

9. Types of Data - There are actually three types of data records that are recorded by the data collection subroutines. The first are records that pertain solely to PRASE. An example of this type is the wrap records for send and receive messages. The second type of record contains data pertaining to some type of Hypercube node call such as copen, probe, recvw, sendw, and syslog. The final record type holds data that the user selects. The user can place a subroutine call into the code and pass a variable of his choosing. Different subroutines are provided to support different variable types. In this way, the user can collect data over time that is of specific interest. Although Seecube will not support presentation of this type of data, other display routines or analysis packages could be developed to analyze the data. It should be noted that Seecube includes an integer field in each instrumentation call that could be used to mark specific calls as unique. In a convoluted fashion, it could also be used to trace integer values.

3.5 Lower Level Design

There are essentially four major parts to PRASE. Each of these parts will be discussed, with some detail being provided. However, the intent here is not to get into every minute detail of implementation. Rather, the purpose of this section is to provide some level of understanding as to how the major parts are designed.

3.5.1 Data Collector The data collector was implemented in two parts. The first part is a set of subroutines, called by both C and Fortran, that perform the necessary functions to collect certain performance data. These subroutines were implemented in C.

Normally, data is collected pertaining to the instrumented call that has just been made. For example, if a user's code calls the PRASESENDW call, the PRASE routine will get a time prior to the service, execute the requested function (*sendw*), get a time after the service, and then store the data.

For the receive without wait this flow may be slightly different. When a user performs this type of receive call, the receive buffer, count, originating node, and originating process id information may be invalid. This can occur if a receive is issued prior to a message arrival. The PRASE subroutine checks to see if the data is valid. If it is, normal flow occurs. If the data is invalid, collection is put on hold until a status call is done that shows that the data is now valid. At this point, the data is stored for the receive. The user is required to perform only one receive without wait for a given channel at a time. Also, the user is required to call status if he does not know that a message is ready at the time the receive routine is called. The user could know that a message was ready prior to issuing a receive without wait by using the probe service provided on the Intel Hypercube. In this case, the receive routine should show a valid status when it checks and all should be well.

Certain data is stored when a *recv* is issued. Included in this data is the time that the original service was called. When the message is finally shown to be received by calling *status*, that original time is recorded in the trace along with an ending time that reflects when the status was good for the message in question. This means that if other trace entries were recorded after the initial call to *recv* but prior to the completion of the event with a good status, those times will not be in order (at least start times for events). If translation to Seecube format is desired, the user must not call another Seecube-supported service during that interval. These would

include *copen, cclose, recv, recvw, send,* and *sendw.* Another call to *recv* might be all right if the user calls it with different parameters, a different channel, and a different message type and insures that he does not call *status* for the second *recv* until the first *recv* completes.

The send without wait service is apparently handled differently in PRASE than it was in Seecube. The Seecube code provided to AFIT appears[3] to require a user to call a separate routine to check status until the send buffer is free. The PRASE code does not concern itself with when the buffer is free. The completion time for the service is considered to be directly after the call is completed. Although it is not unreasonable to require a user to call status to check on his message, PRASE was not implemented in such a way as to hold off recording until a valid status was returned. The user should be aware that the begin and end times do not reflect the same information as when they are recorded for a send and wait message service. Once sends without waits and sends with waits are translated, they will all be displayed as sends starting at the begin time collected and ending at the ending time collected. This could possibly be misleading to the user if he has not been made aware of the situation. Thus, the User's Manual explains the situation. In this case, even though the data does not reflect what it might, it is defined and can be dealt with.

Another unique item should be mentioned pertaining to events that appear to complete in zero time. The data collector, during testing, has been shown to gather data on messages that appear to be sent and received in zero time (or negative time if synchronization was a little off). This apparently happens due to the coarse time slice used by the node operating system (5 milliseconds). When the data is finally displayed using Seecube, if the zero time situation occurred, no data is displayed (except possibly in the data tables provided). This item is reflected in the User's Manual.

[3]The word appears is used here because the author is not totally convinced of his understanding of the Seecube code.

The second portion of the data collection software is a collection routine (program) that should be started in the background on the host prior to running the user's application. This routine will then run until killed by the user. Its job is to collect any data sent to it. The instrumentation code running on the nodes will dump data to this routine which will in turn write the data to a file. Each process running on a node will have a separate file for its data. A file pertaining to node 0 *pid* 1 would be named n0_p1 .

3.5.2 Fortran Preprocessor The Fortran Preprocessor is designed to accept Ryan-McFarland Fortran, and output a new program with instrumentation code added. It determines which calls to change based on what is found in the configuration file. Certain language features in Ryan-McFarland Fortran are not supported. These are listed in the User's Manual included with this thesis.

The main job of the preprocessor is to add code that allows instrumentation to occur. Specific statements need to be added in certain places within the code. The preprocessor must be able to identify its current position within a file. It must know when a change from specification statements to normal code statements occurs. Basically, then, the preprocessor steps through the Fortran input file trying to identify each line's type as it goes.

Each line that is not a comment is cleared of white space characters (blanks and horizontal tabs) and placed into upper case to facilitate searching. Since Fortran is not case sensitive, any combination of upper and lower case letters can be valid. This could turn into a searching nightmare. To remedy this, all letters were changed to upper case. Fortran also allows spaces to be placed freely in the code. Clearing white space is performed to get the line closer to a format that eases the parsing burden.

Another feature of the language had to be dealt with. Fortran allows continuation lines so that long statements can be extended past one physical line in length.

This requires identifying which lines are continued to insure that a line type is not judged incorrectly.

Continuation lines are gathered into a single line which is changed to upper case with white space characters removed. A define statement is used to set a maximum number of lines in which the preprocessor will search for continuation statements. At the initial implementation, this maximum was set to 50 lines. This means that any statement that is continued must be completed within 50 lines or the preprocessing may fail.

The updated code is placed into a file with a slightly modified file name so that the user's original code is left intact. The original file name might be filename.f, which is changed to filename.p.f . The Fortran compiler does not accept a file ending with a p. Thus, this format was chosen for the new file name.

3.5.3 C Preprocessor Like its Fortran counterpart, the C preprocessor was designed to accept code written in C to run on the Intel iPSC Hypercube. It is important to note that the preprocessor is designed to work with code that has already compiled with no errors. Results are unpredictable if syntax errors exist within the code.

The C preprocessor is less complex than the Fortran version. Rather than having to find each call to a subroutine that has been tagged for instrumentation, the preprocessor simply places a define statement at the beginning of the file which then causes all subroutine names referenced throughout the file to be changed to the new name.

An example is in order to clarify what is going on. If a user wants to collect data on all *sendw* Hypercube calls, he places this information in the configuration file. The preprocessor then places a define statement at the beginning of the output file so that all calls to sendw are changed to PRASESENDW. Hence, the instrumentation subroutine is called rather than the normal Hypercube subroutine.

Internally, a dynamically allocated list of lines is kept. Each record of this list contains both the original line and a decommented version of the line. Removing all comments made the preprocessor code much easier to write. Unlike Fortran, C allows executable statements and comments to reside on the same line. A single comment can also span several physical lines. It was much easier having to only parse actual C statements. It should be noted that the C language preprocessor apparently decomments the file as well. This was noticed when the cc -P option was used on the Intel iPSC Hypercube.

The only parsing and changing of code required is within the main routine. To facilitate ease of code implementation, the user is required to place a special comment in the main routine at a specific place. This tells the preprocessor where to add certain code and that it is now in the main routine. It continues to look through the main routine for other instances where code changes are required. Once the end of main is found, its job becomes one of copying the rest of the file into the new output file.

3.5.4 PRASE to Seecube Translator The translator takes data output by the PRASE system and translates it into Seecube format. This translation is done so that the Seecube graphical displays may be used. The mapping of PRASE data to Seecube data has been detailed in Appendix G. Please refer to that appendix to see how the data collected by PRASE can be used to determine Seecube data records.

The translator has been broken into three passes. The initial pass reads in the PRASE data and builds most, but not all of the Seecube fields into new records. Pass 2 fills in all other fields but one. Pass 3 fills in the final field and outputs the data into a file that can be used by the Seecube Resolver. The resolver can be run and the graphical routines can be used to display the data collected.

The translator was originally developed on a DEC Microvax 3 in Ada and was then moved to a Sun workstation using Ada. Because the data file was written in

binary, a problem was encountered at this point. Both the Intel iPSC Hypercube and Microvax order their bytes in reverse order with low byte first. However, the Sun workstation does not. Therefore, the initial reading of the file had to be different depending on which machine was being used to translate the files. Two routines were included in the code, so that, depending on the machine being used, the correct data representation could be achieved.

The PRASE system allows the user to specify a time at which data collection begins. If the user chooses a time other than time 0, all records may not be present for correlation. This might happen if a *sendw* service was executed on a node just before the collection start time, and a *recvw* was executed on the receive node right after the collection start time. The receive data would be captured but the send data would not. When records can not be correlated, the preprocessor will tell the user how many sends and receives were not correlated over all the nodes and processes. Uncorrelated sends and receives can also occur if the maximum number of data records causes collection to stop prior to the end of a run.

The translator will filter out any data records not pertinent to the translation process. The only records of interest to the translator are send type records, receive type records, channel open and close type records, and message count wrap type records. All other record types are ignored in reference to translation.

Another important factor is that only one process was collecting data per node. As far as can be determined, Seecube will not support multiple processes per node. Thus, the translator was not written to support multiple processes running on a single node. If the user's configuration file actually reflects the data collected, and multiple processes per node were used, the translator will flag an error and stop.

Section 3.5.1 discusses certain issues pertaining to sends and receives without waits. These issues are to some degree pertinent to the translator as well. The reader should review that section to gain a better understanding of the translator.

3.6 Items Supported

In Chapter Two, several measurements were discussed. This section will review some of those items and discuss how they are supported in PRASE. Not all items from Chapter Two are implemented in PRASE. Certain measurements are supportable, but are not currently calculated. To come up with specific values for these supportable but not currently calculated measurements, additional software or a statistical analysis package would be required.

3.6.1 CPU Utilization Data is currently captured that can support this measure in one form. If idle time is defined as "the time spent waiting in a *recvw* service while no corresponding send had occurred," then CPU utilization could be calculated from the data collected. The measure, however, is not currently calculated. Seecube is capable of showing when a processor is in a receive state but no send has been initiated.

3.6.2 Turnaround Time A loose interpretation of this measure has been captured within PRASE. Since the start of a run begins at global time zero, the beginning time of a process is known. PRASE saves the last data record for an end record. The end record, then, provides the total run time. This time can then be compared to other runs or just used as is to give the user a feel for the amount of time required by his process.

3.6.3 Seecube Measurements Since translation from PRASE to Seecube format can be accomplished, certain Seecube measures pertaining to message traffic can be supported.

3.6.4 Monit Measurements In a general sense, PRASE can be used to gather information on process activity as well. By judiciously placing marker collection routines in a user's code, one can find out how many times a routine is called or

even how much of the overall time a routine uses. A user might do this by calling the character marker facility upon entering a specific routine. The user could then also call the character marker routine upon exiting that routine. Once this is accomplished, data is present to determine the time spent in a routine. Although this type of process activity does not appear to be what Monit captures, it is in some sense still oriented towards the process rather than the message events.

3.6.5 Local User Measurements of Interest Several measurements were identified by local AFIT users. Many of them are readily available or can be calculated from the data saved. Some of these are message sizes, amounts of messages, number of messages waiting at a given time, message type statistics, queue lengths, waiting time statistics, and speedup (if data from a previous run is also available). CPU idle time and utilization can be determined if the definitions discussed in Section 3.6.1 are used. Waiting time statistics could also be determined pertaining to how long a receive and wait service waited. User markers are supported for several different data types. They include logging the value as well as the time of the logging event.

3.7 Summary

This chapter has accomplished four major items. First, certain project alternatives and the subsequent decisions have been listed. Next, some overall system design considerations were discussed. Third, some details pertaining to each major portion of the PRASE code were provided. Finally, certain items discussed in Chapter Two that are supported/supportable were given. In this thesis' context, supportable means that sufficient data was collected to determine a specific measure, but calculations using that data have not been done.

Hopefully, the reader now has a better understanding of the PRASE system. The next chapter will consider testing issues for PRASE. In the following chapter, and in the User's Manual, problems with the system as well as unsupported items are discussed.

IV. Testing

Testing the PRASE system was approached from three distinct directions. First, as the code was being developed, it was also being tested. Normally, as a subroutine or function was added, the program would be compiled and checked. This was highly unstructured testing but facilitated an initial check of the code and its basic functions.

Second, more structured testing was accomplished for configuration file inputs to the code. The methods used to develop test cases were those of equivalence class testing and boundary value analysis [10]. Both valid and invalid cases were used to test PRASE. This type of testing was also applied to the prase_user.h *include* file set up by the user.

Third, system level testing was accomplished. This type of testing was done to check out system functionality. It also was used to determine how the system fit together as a whole. An example from a preliminary system level test was one where ten messages were sent from one node to another. The receiving node was forced to wait several seconds before accepting the messages. Then, ten receives were accomplished. The purpose of this test was to insure that the system as a whole was doing what it was supposed to be doing. The results were viewed on the Seecube graphical displays and in fact reflected the expected algorithm performance.

The remainder of this chapter is devoted to these last two types of testing. In the next section, the equivalence classes and boundary values will be discussed. In the following section, an overview of the system level tests will be given to show the types of system functions tested. For a breakout of the actual test cases and results for both of these categories, the reader should refer to Appendix D. An overview of these results is included in Section 4.3 of this chapter.

4.1 Equivalence Classes and Boundary Values

A major means of input to the PRASE system is through the configuration file. Therefore, most of the tests developed via equivalence classes pertain to the configuration file in some way. As the equivalence classes are listed, boundary values will be used in specifying how a class should be tested.

The user also has the means of tailoring the system by changing values in a user include file. The values from this file will be considered in section 4.1.2 .

4.1.1 For the Configuration The configuration file is structured in a very specific way. For an example of this, refer to Appendix H. The purpose here is to identify both acceptable and unacceptable inputs from which actual test cases can be generated. No attempt has been made to cover all possible cases of errors and valid combinations; rather, the purpose is to test the general rules allowed in making up a configuration file. The preprocessors handle configuration file inputs in a different way than the translator. Thus, any test cases generated should be run against both preprocessors as well as the translator.

Six different groups of information are allowed in the configuration file. Each of these groups will be discussed. Comment lines will also be considered. As a convention, the equivalence class number will be placed in parenthesis following the text for each class.

- Comments - A comment line should begin in column one with a '#' sign. Anything following on the line is ignored.
  - Valid Equiv. Class - Begin a comment with a '#' in column 1. (1)
  - Invalid Equiv. Class - Begin a comment with a '#' in column 2. (2)
  - Invalid Equiv. Class - Include a comment but with no '#' sign. (3)
- Node Information Groups - These groups hold information pertaining to which nodes within the Hypercube are actually running code. Node groups and pid

groups must go together with a pid group being the very next group encountered in the file. A node group consists of the group identifier on the first line, a beginning node on the next line, and the ending node on the third line. For more specific information about this group, refer to Appendix H.

- Valid Equiv. Class - Include a valid Node group in the file with no other errors. Have the beginning node be the lowest node number possible and the ending node the highest node number possible. (4)
- Invalid Equiv. Class - Place the group identifier (NODE) beginning in column 2. (5)
- Invalid Equiv. Class - Leave out the beginning node information. (6)
- Invalid Equiv. Class - Leave out the ending node information. (7)
- Invalid Equiv. Class - Provide an invalid range for the beginning node information. Use a -1 for the value. (8)
- Invalid Equiv. Class - Provide an invalid range for the ending node information. Use a value one greater than the largest possible node number. (9)
- Invalid Equiv. Class - Place the beginning and ending node information into the file in reverse order. (10)

• Pid Information Groups - A pid information group lists the pid numbers of the processes that will be running on the nodes listed in the previous node group. A pid group must contain at least three lines but can have up to four lines. The first line must hold the group identifier beginning in column 1. The next line should contain a single number representing the number of pids that will be listed. The third line holds actual pid numbers separated by at least one space. Up to ten pids can be listed on line three of a pid group. If more than 10 pids are required, the first 10 MUST go on line 3 with the remainder of the pids being listed on line 4.

- Valid Equiv. Class - Include a pid group with a minimum number of pids. Use the lowest possible valid pid number. (11)
- Valid Equiv. Class - Include a pid group with the maximum number of pids. Insure that at least one of the pids is the largest number a pid can be. (12)
- Invalid Equiv. Class - Place the group identifier in column 2. (13)
- Invalid Equiv. Class - Leave out the group identifier. (14)
- Invalid Equiv. Class - Insure that a node group is included but do not include a pid group. (15)
- Invalid Equiv. Class - Include a pid group prior to a node group. (16)
- Invalid Equiv. Class - Leave out line 2 of the pid group. (17)
- Invalid Equiv. Class - Leave out line 3 of the pid group. (18)
- Invalid Equiv. Class - Include a pid number that is one number too small for the valid range of pid numbers. (19)
- Invalid Equiv. Class - Include a pid number that is one number too large for the valid range of pid numbers. (20)
- Invalid Equiv. Class - Use more than 10 pids but only place 8 on the third line of the pid group. Place the remainder of these pids on the optional fourth line. (21)
- Invalid Equiv. Class - Use a number of pids of -1. (22)
- Invalid Equiv. Class - Use a number of pids one greater than the valid range. (23)

• Instrumentation Information Groups - This group communicates to the PRASE system which calls to instrument. Any calls listed will cause data pertaining to that specific call to be gathered. The group consists of a group identifier in column one of line one, an integer value on line two representing the number

of calls to instrument, and subsequent lines containing one name per line of the calls to instrument.

  - Valid Equiv. Class - Include a valid group with all the valid calls being instrumented. (24)
  - Valid Equiv. Class - Include a valid group with only one call being instrumented. (25)
  - Valid Equiv. Class - Include a call to instrument. Do not start it in column 1. (30)
  - Invalid Equiv. Class - Begin the group identifier in column 2. (26)
  - Invalid Equiv. Class - Leave out the group identifier. (27)
  - Invalid Equiv. Class - Include a number of calls of 0. (28)
  - Invalid Equiv. Class - Include a number of calls that is one larger than the maximum. (29)
  - Invalid Equiv. Class - Include an invalid instrument type (all in capital letters) in the proper location. (31)
  - Invalid Equiv. Class - Include two valid instrument types next to each other on the same line. (32)

- Libraries Information Group - This group is for future use and is not required to be in the configuration file. However, it should be accepted as valid if included. The future purpose is to support automatic makefile generation. The user could stipulate any libraries or object modules he would like to add to the defaults already built into the system. The group must begin with the group identifier in column one of line one. The next line includes the number of libraries/object modules the user wishes to add. Subsequent lines hold the user entered values.

- Valid Equiv. Class - Include a valid group with the minimum number of entries allowed. (33)
- Valid Equiv. Class - Include a valid group with the maximum number of entries allowed. (34)
- Valid Equiv. Class - Include a number of entries of 0. (37)
- Invalid Equiv. Class - Place the group identifier in column 2. (35)
- Invalid Equiv. Class - Leave off the group identifier. (36)
- Invalid Equiv. Class - Include a number of entries one greater than the maximum allowed. (38)
- Invalid Equiv. Class - Place two library/object names on the same line. (39)

• Start Time Information Group - This group includes the time that the collection software should start collecting data. The first line of this group should include the group identifier starting in column one. The next line should include the start time.

- Valid Equiv. Class - Include a valid start time group with the smallest start time possible. (40)
- Invalid Equiv. Class - Place the group identifier starting in column 2. (41)
- Invalid Equiv. Class - Leave out the group identifier. (42)
- Invalid Equiv. Class - Leave out the time. (43)
- Invalid Equiv. Class - Include a start time of -1. (44)

• File Names Information Group - This group holds the names of the files that the user wants PRASE to preprocess. The group begins with a group identifier on line one and starting in column one. The second line holds the number of files to process. Subsequent lines hold the actual file names.

- Valid Equiv. Class - Include a valid group with the minimum number of files allowed. (45)
- Valid Equiv. Class - Include a valid group with the maximum number of files allowed. (46)
- Invalid Equiv. Class - Begin the group identifier in column 2. (47)
- Invalid Equiv. Class - Leave out the group identifier. (48)
- Invalid Equiv. Class - Include a number of files of 0. (49)
- Invalid Equiv. Class - Include a number of files that is one larger than the maximum. (50)
- Invalid Equiv. Class - Include a file name that does not start in column 1. (51)
- Invalid Equiv. Class - Include a file name that does not exist. (52)
- Invalid Equiv. Class - Include two file names on the same line. (53)

• Invalid Group Identifiers - Group identifiers are required to be in capital letters and begin in column one. There are only six valid identifiers.

- Valid Equiv. Class - Include all six valid identifiers. (54)
- Invalid Equiv. Class - Include the group identifiers with some lower case letters. Do this for each identifier. (55)
- Invalid Equiv. Class - Include a group identifier name that is all in capital letters but is not one of the six valid ones. (56)
- Invalid Equiv. Class - Include two group identifiers on the same line. (57)

• Others - A few miscellaneous tests will be listed here.

- Invalid Equiv. Class - Run the preprocessors and translator with no configuration file present. (58)

- Invalid Equiv. Class - Run the translator with a missing data file. For example; if the configuration file says that there was a node 0 pid 0 process running, then there should be a file containing the data gathered. That file should be present at translation time. Remove that file and run the translator. (59)

4.1.2 User Include File Inputs A user can change values in a user *include* file that tailors the system to his application. Four values are currently required in this file. First, a value should be provided that determines the maximum number of data records stored for a particular application. Second, a value should be included that stipulates how often dumping occurs. Third, the user should provide a value that is greater than or equal to the number of message types used in any particular application. Finally, the user needs to provide the maximum number of *pids* being run on any one node. The principals of equivalence classes and boundary value analysis will be applied in most cases to the values found in this file [10]. The *include* file discussed here is the prase_user.h file. Another *include* file called PRASE_FOR.H (for Fortran use only), has one value that must be the same as a value in the prase_user.h file. No official tests were accomplished for this file.

- Maximum Number of Data Records - This value is used to set up the maximum number of records that will be collected for the associated application.
  - Valid Equiv. Class - Include a valid number. (60)
  - Invalid Equiv. Class - Include a negative value. (61)
- Maximum Number of Records Saved Prior to Dump - This value is used to determine how often the collection routines should dump their data.
  - Valid Equiv. Class - Include the lowest valid number here. (62)
  - Valid Equiv. Class - Include a number larger than 1000. (63)

- Invalid Equiv. Class - Include a 0 here. (64)
- Invalid Equiv. Class - Include a number that is too large. (65)

- Maximum Number of Message Types - This number is used to stipulate the maximum number of message types allowed for a specific application.
  - Valid Equiv. Class - Include a number the same as the number of message types used. (66)
  - Valid Equiv. Class - Include a number larger than the number of message types used. (67)
  - Invalid Equiv. Class - Include a number smaller than the number of message types used. (68)
  - Invalid Equiv. Class - Include a number of 0. (69)

- Maximum Number of *pids* per Node - This number is used to stipulate the maximum number of processes running on any single node in the system.
  - Valid Equiv. Class - Include a number the same as the number of *pids* being used. (70)
  - Valid Equiv. Class - Include a number bigger than the number of *pids* being used. (71)
  - Invalid Equiv. Class - Include a number smaller than what we are using. (72)

4.2 System Level Testing

This type of testing has less structure in the formulation of what to test than did the equivalence class/boundary value analysis type of testing. It is, however, just as important. Its purpose is to test functions of the system as well as the system

as a whole. The question one might raise is, "Can the system, from start to finish, accept a user's code and present the correct results?"

Decisions had to be made as to what areas to test and how to test them. Trying to test everything possible, or even every function, was not even considered. Rather, several tests were chosen to "try out the system" in an attempt to provide the reader with some level of confidence in its capabilities.

The best test for a system of this kind is use by multiple users. This stems from a philosophy that live data is an excellent testing aid. The more use over time the system gets, the more problems that can be identified and fixed. However, since time is not always in great supply, the following test cases were used.

- Test Case 1 - This test's goal is to determine if the system captures the proper flow and timing of messages. The following items were incorporated into the test.

    1. Send 10 messages, one every second from one node to another.
    2. Have a second node wait 20 seconds and receive messages one at a time every second.
    3. Use the clump and queue displays as well as the time provided by Seecube to insure that messages are being displayed in the correct amounts at the correct times.

- Test Case 2 - This test is designed to check communication paths shown by Seecube. Correct communication paths will be determined based on the deterministic Intel routing algorithm.

    1. Send a message from a node to a nearest neighbor node. Insure that the path is correct.

2. Send a message from a node to another node that has one intermediate node. Send the message back. Insure that the paths are correct. The path back should be different.

3. Send a message from a node to another node that has at least two intermediate nodes. Send the message back. Check the paths.

- Test Case 3 - This case tests to see if data gathering starts on or after the time provided in the configuration file.

  1. Send 1000 messages from one node to some other node(s).

  2. Provide a start time in the configuration file that should be a time somewhere in the middle of the time required to process these messages.

  3. Check the data gathered to insure that data begins being collected at the correct time and that the data displayed is correct.

- Test Case 4 - The purpose for this test case is to check for validity of global times being saved in the traces. The data gathered in Test Case number 3 should be used for the analysis.

  1. Times for corresponding sends and receives will be considered to check for validity. This is very subjective but is important none the less.

- Test Case 5 - This test case should be used to test the translator in conjunction with the data collector.

  1. Use the data collector to capture several types of records.

  2. Run the results through the translator to insure that, even with data that is not pertinent to translation, the translator still works.

- Test Case 6 - This test case uses code prepared by someone other than the author to test the preprocessing function.

1. Obtain and preprocess a Fortran application and check the results.
2. Obtain and preprocess a C application and check the results.

- Test Case 7 - This test case will be used to test the user callable marker routines provided by PRASE.

  1. Use a C program to write to each allowed marker routine and then check for a correct trace (with correct values).
  2. Use a Fortran program to write to each allowed marker routine and then check for a correct trace (with correct values).

- Test Case 8 - This test case will exercise the multiple processes per node option allowed by PRASE.

  1. Use a program that runs multiple processes per node.
  2. Include known communication patterns between nodes and between processes on the same nodes.
  3. Collect data and analyze the results.

- Test Case 9 - The purpose of this test case is to "try out" the system as a whole.

  1. Use the local Hypersim libraries and a driver routine.
  2. Take the example all the way from the code libraries to displaying the results using Seecube.
  3. Analyze results.

4.3 Evaluation of Test Results

Appendix D contains information describing how to obtain a copy of the actual test cases and an evaluation of the results for each case. Accompanying the results, any problems identified are provided along with the action (if any) taken.

Here, an overview of the results is provided.

- In general, configuration file and prase_user.h inputs are not checked as well as they should be or are not handled properly when errors are induced (possibly causing erroneous results). Several tests brought out the fact that more explicit error messages would be helpful when dealing with user inputs into the configuration file and prase_user.h file. At this point, the User's Manual was the vehicle chosen to deal with this problem. It tells the user how to make up the configuration and prase_user.h files properly. Specific problems have been listed in Test Results (all errors will not be listed here). At some point in the future, it would be nice to handle these problems in a more user friendly way.
- A few error messages in the code were either updated or rewritten to better reflect the situations encountered.
- At least in one case, code updates were performed to fix a problem.
- In the case of the translator, certain errors were ignored because it does not need to deal with some areas of the configuration file. In some cases, this caused the translator to run to completion without looking for any files to process (because of errors in certain groups). Since the translator will probably not be used without first using a preprocessor, this was considered acceptable.

  More checking and more error messages would be nice; however, they will not be accomplished at this time. The main vehicle for dealing with this problem was to insure that the User's Manual has information pertaining to the correct formats for these files.
- When the maximum number of message types is smaller than the number of types a user attempts to use, there is a problem. The only indication of the error is in the trace itself. A value of -99 is placed in the trace in the count field. This was by design. However, if a user does not examine the trace, and just processes the data, he is likely to get erroneous results. The User's Manual

is the current vehicle for dealing with this problem. However, eventually an error message should be written to the *syslog* file so that the user will have some indication of a problem.

- Certain erroneous entries in the prase_user.h file are flagged at compilation or link time. Correct ranges of inputs must be entered to be able to compile. The User's Manual is again the means used to tell the user what the appropriate entries should be.

- When instrumenting *syslog* calls, a problem was noticed. The following is an example of code that would cause the problem.

  CALL SYSLOG(mypid(),'my msg')

  The preprocessor changes the call by assigning the character portion of the call to a temporary variable and then passes that variable into the instrumented version. The problem was that the character version placed into the temporary variable is changed from the original. It was changed to uppercase and all blanks were gone. The User's Manual reflects this problem. To get around this, the message can be assigned to a variable prior to calling *syslog*. Then *syslog* can be called with the variable name rather than a character constant. This problem only pertains to the Fortran preprocessor. No fix will be accomplished at the current time.

- Apparent problems with the Memory Cube were encountered. Problems that were experienced, after a reboot, were not repeated.

- A problem was uncovered in the procedures being used. The maximum *pids* value was being changed in the prase_user.h file but not in the PRASE_FOR.H file. In certain situations, this appears to be a real problem. The User's Manual reflects the need to keep these straight.

- An operational item pertaining to Seecube was found in that unless sliders are set correctly, the trace will not play all the way to the end without user

interaction. The correct interaction, and the fact that this item exists, were documented in the User's Manual.

- The following Fortran statement type was found to be unsupported.

```
200         if(status(HOSTCHAN).eq.NOTBUSY) goto 300
```

  The preprocessor did not instrument the status call as it should. This was not considered in the design of the preprocessor. The User's Manual reflects the fact that this is an unsupported feature allowed by Fortran.

- Certain host code used in testing stopped the processes on the nodes by killing them. Thus, the PRASEEND routine was never encountered and no data was collected. The User's Manual discusses this situation.

Certain unsupported features are known to exist. For example, logical *if* statements in Fortran are not supported when a call to a routine the user wishes to instrument is found in the result portion of the *if* statement. Statement function statements in the main routine of a Fortran program are also not supported. Also, for a user to reconfigure the run he wishes to instrument, he must begin again at the preprocessing stage rather than just being able to run code on different nodes and *pids*.

Although testing brings out negative aspects of the project, positive aspects were also realized. In certain cases, things seemed to go well. Apparently correct results were able to be captured and displayed. Even though several errors have been uncovered, it appears that PRASE can be used to aid a parallel programmer.

4.4 Summary

This chapter has attempted to convey the testing that applies to the PRASE system. When changes are made to the system in the future, it would behoove the maintainer to rerun pertinent test cases to insure that proper system functionality is maintained. As the system matures, test cases should be added or deleted from this list as appropriate.

V. Conclusions and Recommendations

5.1 Conclusions

It is exciting to have provided a new monitoring capability to the AFIT user community. However, this capability is not as mature as it should be. First, PRASE is not yet easy enough to use. If a system is hard to use, it will probably not be used. This chapter contains some recommendations pertaining to this aspect of PRASE. Second, PRASE needs many and varied user applications to test it out thoroughly. Something on the order of Beta testing is needed to really exercise the system. The PRASE system delivered with this thesis provides a foundation for further work. It is useful and may prove valuable to several users.

Overall, even with many and varied problems, the effort seems to be an initial success. PRASE can monitor code on the Hypercube. It has maintained compatibility with Seecube, thus providing the important graphical displays. PRASE is a foundation that can be built upon in further work.

5.2 Recommendations

PRASE has only begun to approach what it should become. As mentioned in Chapter One, PRASE is not yet a software environment. Much work could be done to make the system more usable as a whole. The following is a list of recommendations with a short explanation where needed.

- Graphical Interface - At least two functions in a graphical interface seem to be of some possible use. First, a user interface that ties several of the PRASE programs together would be very helpful. A user then might only need to press buttons to initiate commands that currently must be typed. The interface could also guide a user through a monitoring session by presenting a step by

step list of items to be accomplished. The configuration file could be built by the interface alleviating potential problems with bad configuration files.

Second, new graphical techniques could be explored in presenting the data collected. Three dimensional representations are of particular interest to this author.

- Preprocessor Enhancements - The preprocessors could be extended to include an automatic makefile generation capability. This would add any user specified libraries/object modules to a default set of required libraries/object modules to produce a makefile. This might be another item that would make the monitoring task easier for the user.

  There seems to be some value in extending the Fortran preprocessor to support more of Ryan-McFarland Fortran. At this point, it does not totally support the language. The C preprocessor could also be made more general by releasing the user from the need to place a 'special' comment in his code.

  Another idea pertaining to updating the preprocessor is to support data collection on subroutine utilization. If placed in a subroutine monitoring mode, the preprocessor could automatically place the appropriate calls in the code to record start and stop times for each subroutine. The data could then be used to determine which subroutines were taking the most time.

  Another mode the preprocessor might have would be one of a variable monitoring mode. It would be useful if it could be told the name of a variable to trace, and the preprocessor would automatically set the code up.

- Data Collection Enhancements - These routines could be extended to support vector routines, or any routines of interest to a user.

  Also, the data collection software should be updated to be more flexible in allowing the user to run his code on different node configurations without having to go through the whole process each time a change is made. A potential

idea in this area is to let the prase_clct routine upload the configuration to the node code as opposed to compiling it in.

The collector should put out error messages when wrap count errors are possible. Rather than just writing a designation to the trace, a write to *syslog* should also be accomplished.

The situation with the start times pertaining to a *recv* service should be fixed. Briefly, this is where a *recv* can be called and before it can be shown to complete, other trace records are written. The problem occurs in that the start time recorded, once a *status* clears the *recv*, is possibly earlier in time then the previous item in the trace. This means that the trace no longer insures a time order sequence for all times entered. Seecube was written to deal with this problem by having separate events for non-blocking services [4:28]. This enhancement should be made and the translator updated. This will hopefully make it harder for a user to be able to collect bad data. The growth of the system should definitely be directed toward protecting the user.

- Timing Studies - It would be interesting to know the overhead placed on the system when PRASE was invoked. Timing studies for diffcrent situations could be accomplished.
- Module Sizing - Another valuable piece of information to know would be how big the code you are adding is. This could be done based on which subroutines were being called as well as how many records, *pids*, and message types were being used.
- Testing - Several users making use of the code would be valuable in a testing sense. The use could bring to light other problems or enchancements that might be valuable.
- Error Corrections - Several problems have been listed in Chapter 4. These problems should be resolved.

5.3 Summary

Admittedly, PRASE still has several problems to overcome. It is not as user friendly as it ought to be and certainly does not provide all the capabilities it might. Graphical shells, user directed processing, better data collection capabilities and more could make the system much better.

PRASE has just begun. If it is enhanced, it can become a more useful tool for the parallel processing user. I hope that the effort will not stop here.

Appendix A. *User's Manual*

The following pages contain the User's Manual for the PRASE system. Pages are numbered based on the manual, not on this appendix.

PRASE
User's Manual

December, 1988

Table of Contents

List of Figures

PRASE

User's Manual

I. Introduction

1.1 Overview

PRASE is an instrumentation software package that allows user of the Intel iPSC Hypercube to gather data pertaining to software that executes on the Hypercube nodes. A user can gather data pertaining to message passing events as well as other Hypercube specific subroutine calls. Facilities also exist to allow a user to monitor variables within his program. This gives the user the ability to keep track of a specific variable's change over time.

Synchronization is performed so that events happening on one node can be compared, in some rough sense, to events happening on another node. PRASE is compatible with Seecube, a software package obtained from Tufts University in Massachusetts. The Seecube Sequencer can be used to graphically display message passing data collected using PRASE.

Graphical displays do not exist for all of the features made available through the use of the PRASE data collection software. Also, several steps are required to use PRASE. PRASE may be a tool that will aid you in a better understanding of your parallel processing software. Comments on how to make the system better are welcomed; because PRASE is a tool for the user.

1.2 About this Manual

The next five chapters will walk you through the step by step requirements for using PRASE. Certain setup procedures are required on the machines you will

be using. Once set up, you will use one of two preprocessors to prepare your code for use with PRASE. Once you have completed preprocessing, it will be time to collect data on the Hypercube. Next, if you wish to use Seecube to graphically view message passing results, you will use a translation program to prepare the data for use by Seecube. The Seecube Resolver should then be used, followed by the Seecube Sequencer.

After the step by step procedures have been given, a chapter has been included that discusses how a you can collect data on your own variables and program execution. This chapter includes syntax.

The last part of this manual touches on some very important items that you as a user should know. To use PRASE effectively, and to better understand what PRASE is doing or isn't doing, you should read the *entire* manual (especially the last part). If you don't read the entire manual, you may assume that PRASE is doing something that it is in fact not doing.

1.3 Odds and Ends

You should have accounts on three machines if you want to be able to take advantage of the full range of capabilities provided by PRASE (and Seecube if you wish to use it). The three machines you will need access to are (1) any color Sun Workstation, (2) any Microvax running Digital Equipment Corporation's Ultrix operating system[1], and (3) an Intel iPSC Hypercube.

The code you want to gather data on should be written in one of two languages. It can be in C for the iPSC Hypercube or in Ryan-McFarland Fortran for the iPSC Hypercube. One more thing about your program before we proceed. Your code must compile with no errors before you use the preprocessors. If it does not compile, you should not attempt to preprocess your code. More on this later.

[1]The translator and preprocessors will run on a Microvax under the Ultrix system. The Seecube software must run on a Sun Workstation.

1.4 Using PRASE

An example is included here to give you a feel for the steps required to use PRASE.

> Peter Parallel is a new hot-shot parallel programming whiz who just arrived at AFIT. He has written his first parallel program (and possibly his last if his instructor gets a look at his code). He even has gotten his code to compile and run on the Hypercube but cannot understand why he is not getting superlinear speedup over the sequential version. Needless to say, he thinks it is a hardware problem.
>
> He talks to the system manager about running diagnostics, who calmly suggests that he might have some unexpected software overhead. Well, Peter is hurt; but, with great restraint he accepts the challenge of monitoring his code behavior using a package called PRASE.
>
> Peter is told that he can either use a Microvax or a Sun Workstation to prepare his code for monitoring. He chooses the Microvax and begins. First, he copies his source code (which compiles) to the Microvax. He then runs a script file which sets up certain commands he will need in the course of using the Microvax portion of the PRASE system. He then executes his first command to copy files into his default directory. He needs to do this the first time he sets things up for a new program he wishes to monitor.
>
> Next, Peter edits a configuration file that communicates to PRASE certain required information pertaining to his software and what exactly he desires to monitor. Once editing is complete (and Peter figures out how to get out of the editor), he runs the preprocessor for C Hypercube code (since C is the language his program is written in). He then copies the new version of his source code back to the Hypercube.
>
> Now, Peter logs on to the Hypercube and runs a setup procedure there. Again, since this is the first time he is using PRASE for this program, he must execute a script that sets things up for new usage. Peter needs to edit a user file to prepare PRASE (and again he has trouble getting out of the editor). Once complete, he compiles the PRASE subroutines using a previously set up command.
>
> Peter is almost ready to link in the PRASE code with his preprocessed code. He changes his makefile so that the PRASE subroutine's object module gets linked in, and he compiles and links his code.
>
> Peter now gets the Hypercube for his use and loads it for the proper dimensions. Next, he uses another command set up for him to begin a

collection routine that runs in the background on the host of the Hypercube. Once done, he runs his code. Once complete, Peter now must move the data to a Sun Workstation since he wants to use Seecube to look at the results. He does this (using the binary mode in ftp).

Peter again runs a setup command and then executes the translator to prepare his data for Seecube. After translation, he uses the Seecube Resolver and then the Sequencer. Just then, the system manager walks over and happens to glance at the screen. He mentions to Peter that it appears that he is sending ten messages every time any communication takes place. Peter proudly replies, "you bet, I wanted to make sure my messages didn't get lost."

Although none of us are like Peter Parallel (except for maybe the author and developer of the system), we often make mistakes without realizing what we are doing. Hopefully, by following the steps illustrated in our example, you will be able to use PRASE (and Seecube) to gain a better understanding of what your software is doing.

The following figure contains a summary of what needs to be done to use PRASE from start to finish (including using Seecube). You may want to use it as a reference or checklist to remind you of what steps are required. Several commands and acronyms are included that will be explained later in the manual. During your initial reading, use this figure as an overview for what is required. Later, once you become proficient at using PRASE, use this figure as a checklist.

With all of this in mind, let's get started.

```
Execute the following on either a Sun or Microvax

   Setup the commands needed (psun or pvax)
   Make a new directory (For an application never instrumented)
   Get the data needed (For new application - prase_new)
   Edit the Configuration file (If new)
   Copy in the code to instrument
   If C code - add special comment
   Preprocess code (prase_c_pre or prase_for_pre)
   Move preprocessed code to the Hypercube

Execute the following on an Intel iPSC/1 Hypercube

   Make a new directory (For an application never instrumented)
   Setup the commands needed (pset)
   Get the data needed (For new application - prase_newcube)
   Edit the prase_user.h file (If needed)
   Edit the PRASE_FOR.H file (If a Fortran user - If needed)
   Compile your copy of PRASE (pcode_compile)
   Compile (not link) your code
   Add any needed statements to your makefile
   Link you code
   Get the Hypercube (getcube)
   Set the log file to your own file (cubelog -l mylog)
   Clear the Hypercube for your run
   Start the background collection routine (prase_clct)
   Run your program
   Once complete - check the data in prase_data (use pv if needed)
   Check your copy of the log file
   Move the data back to the Sun/Microvax (use binary transfer)
   Find the pid of the background process and kill it
   Release the Hypercube

Execute the following on either a Sun or Microvax

   Setup the commands needed (psun or pvax)
   Change default to the appropriate directory
   If you want to analyze the data on your own, go do that now
              disregarding the following steps.
   Translate the data into Seecube format (prase_trans)

Execute the following only on a Sun

   Run the Seecube Resolver (resolve)
   Run the Seecube Display program (seecube)
   Analyze results
```

Figure 1.1. User Checklist

II. Set Up

Once you have computer accounts, and code that runs on a Hypercube, you may wish to find out more about your program's behavior. When you decide to use PRASE as an aid to this end, you must begin with some set up steps. The following steps will set up certain commands that you will need to use PRASE. Since there are three machines that can be used with PRASE, there are three set up procedures that you need to know about.

2.1 Sun Workstation

In your login file, you should include the following command:

```
alias psun 'source /usr/PRASE/bin/prase_sunsetup'
```

Whenever you log onto a Sun to use PRASE, you should then type the command **psun** to set up other commands that you might need.

If you are beginning to use PRASE for a program that has never been instrumented, you should set up a new directory to hold the files you will need. Once the directory is established, execute the following commands:

```
% cd newdirectory
% prase_new
```

The **prase_new** command will copy in certain files you will need later on. After you have executed the command, do a directory to get a feel for what has been copied in. At some point prior to running a preprocessor or the translator, you must set up the configuration file that was copied into the directory. See the section in this chapter pertaining to this file.

2.2 *Microvax*

In your login file, you should include the following command:

```
alias pvax 'source /usr/PRASE/bin/prase_vaxsetup'
```

Whenever you log onto a Microvax to use PRASE, you should then type the command **pvax** to set up other commands that you might need.

If you are beginning to use PRASE for a program that has never been instrumented, you should set up a new directory to hold the files you will need. Note that if you are on a system that shares (yellow pages) disks between machines, and you have already done the following on a Sun Workstation, you should just be able to change your default directory to the new directory that has already been established. If, however, there is no new directory established, make that directory and execute the following commands:

```
% cd newdirectory
% prase_new
```

The **prase_new** command will copy in certain files you will need later on. After you have executed the command, initiate a directory command to get a feel for what has been copied in. At some point prior to running a preprocessor or the translator, you must set up the configuration file that was copied into the directory. See the section in this chapter pertaining to this file.

2.3 *Intel iPSC Hypercube*

In your login file, you should include the following command:

```
alias pset 'source /usr/PRASE/bin/prase_setup'
```

Whenever you log onto a Hypercube to use PRASE, you should then type the command **pset** to set up other commands that you might need.

If you are beginning to use PRASE for a program that has never been instrumented, you should set up a new directory to hold the files you will need. Once the directory is established, execute the following commands:

```
% cd newdirectory
% prase_newcube
```

The **prase_newcube** command will copy in certain files you will need later on. It also creates a directory to hold all data that gets collected. This directory is called *prase_data*. After you have executed the command, do a directory to get a feel for what has been copied in.

One file copied is the prase_user.h file. This must be set up by you prior to using the PRASE collection software. See the section in this chapter pertaining to this file. If you are a Fortran user, you will also need to setup the PRASE_FOR.H file. For more details on this file, see the pertinent section in this chapter.

2.4 The Configuration File

The preprocessors and translator require a configuration file to tell them certain things about a program. The configuration file MUST be named prase.cfg . At this time, there are five groups of information that should be included in this file. For a specific explanation of this file and how it is formatted, see the example file found in Appendix A.

Here, an overview of these five groups of information will be provided to aid in an understanding of the file's use. Before discussing the five groups, however, comments in the file need to be addressed. Comments may be included by placing a # sign in column one of any line. Anything after the # sign is ignored. Lines should

be limited to 80 characters per line. When you execute the new setup command, a basic configuration file is copied into your directory. All you should have to do is change some information to reflect your configuration.

PRASE needs to know which physical nodes on the Hypercube will be running your program. It also needs to know what the process id (*pid*) numbers will be. To convey this information to PRASE, you must enter **NODE** and **PID** groups into the configuration file. *You should only enter a node number once.* Specific details on how to enter your node numbers and *pid* numbers are given in the attachment by way of comments in an actual configuration file. PRASE will support multiple processes per node, however, if you choose to run in this mode, you will not be able to translate the results into Seecube format.

WARNING

If you leave out the **PID** group(s), the preprocessors will not flag an error. They will however, set up the run as if no processes were running and results will be erroneous if the code will even run in this configuration. Make sure that the **PID** group(s) are present and correct. Also, using the example configuration file and directions therein, insure that all groups are correctly formatted.

WARNING

You must insure that the **PID** group(s) contain *pids* that are numbers in the appropriate range. Negative values are not allowed but will be accepted by the preprocessor. Be very careful in building this group. Also, remember that certain *pids* are used by PRASE. A valid range for a *pid* as given in the Intel documentation is from 0 to 32767. However, 32767 is reserved for use by PRASE. If possible, it would be best to only use *pids* with numbers below 32760.

A start time (**STARTTIME**) group should also be included. This allows a you to tell PRASE at what time to start collecting general data. This can be useful if you only wish to analyze a certain part of your program. It could also be useful if

you can not collect all the data for a run (if the amount of data would just be too much). You must enter an integer number representing the amount of milliseconds after a run begins that you want PRASE to begin collection.

Another group is the **INSTRUMENT** group. This is where you tell PRASE for which Hypercube calls to collect data. The calls for which you can collect data are: *cclose, copen, flick, greenled, probe, recv, recvw, redled, send, sendw, status,* and *syslog.* Although you can collect data on all of these calls, if you plan to use Seecube, the only calls that pertain are *cclose, copen, recv, recvw, send, sendw,* and *status.*

CAUTION

IF YOU PLAN TO COLLECT DATA ON THE *recv* HYPERCUBE CALL, YOU MUST COLLECT DATA ON *status* CALLS. You must also only allow one *recv* per channel or message type to be outstanding at a time and you MUST call *status* until a good status is received. The only exception to this rule is if you have called *probe* prior to calling the *recv* service and that you know that a message is pending when you call *recv.* If you don't follow these rules, data can get confused and receives may not show up in the data trace or they may have bad information pertaining to correlation.

A final group you should know about is the **OLDFILENAME** group. This is wh[illegible] you enter the filenames of the files containing the code you want to instrument. PRASE allows up to 250 files to be preprocessed at once. If you have all of your subroutines in separate files, this should allow you to put all (or most) of them in one configuration file. The file names you use should be NO longer than 12 characters including the extension.

CAUTION

Make sure that the file names are not placed on the same physical line. If they are, certain names may get ignored or you may experience an error when running a

preprocessor. Again, refer to the provided configuration file for the proper format of this group.

WARNING

PRASE will produce a preprocessed file with a slightly different name from the original. However, if a file already exists with the same name as the new name being written, the old file will be destroyed. For more information pertaining to the new file names, refer to Chapter III of this manual.

In way of summary, you need to tell PRASE about these different things required in the configuration file. By providing this information, the preprocessors and translator are then able to handle your programs and data in an appropriate manner.

2.5 The prase_user.h File

As a user of the PRASE data collection routines, you are allowed to tailor the system to fit your needs. An example of the default prase_user.h file that gets copied into your directory when you perform set up procedures on the Hypercube is attached. This file contains comments that will help you set up your file. A brief overview will be given here.

There are four items that you may set up in this file. You must tell the system the maximum number of data records to collect (PRASE_TOTREC_THRESHOLD). This can be very large if you so choose. You must also stipulate the maximum number of records PRASE should save in memory prior to dumping data to the host (PRASE_DUMP_THRESHOLD). PRASE will save data up to this threshold value and then dump that data and begin saving again. Thus, these first two values can be used in conjunction to provide several types of monitoring environments. If your program is aborting, you can tell PRASE to dump after every record (which will hopefully give you some clues as to the problem). This distorts the behavior of your

program greatly but may help to isolate a problem. If the two numbers you enter in the file are equal, then all your data will be saved in memory so that dumps do not distort the program as much. If a you want to collect lots of data, you can also adjust the numbers to save quite a bit of data in memory with dumps occurring periodically.

CAUTION

If you enter a negative value for the maximum number of records to save (PRASE_TOTREC_THRESHOLD), no data will be collected.

WARNING

If you enter a value less than or equal to 0 for the maximum number of records saved prior to a dump (PRASE_DUMP_THRESHOLD), you will most likely get an error when you attempt to compile the prase subroutines using the **pcode_compile** command explained later in this manual. Also, if the value entered is too large, you will also get an error message. The value for 'too large' depends on the size of your program.

So as not to use up too much memory, you are asked to provide two more values. These values are used in determining array sizes. You should enter a value for the maximum number of message types referenced by your program. If a program sends and receives seven different message types, then the number seven or higher should be entered. It is a good idea to make the number somewhat larger just in case you count incorrectly. However, making the number too large takes away memory from your program. You must also enter the maximum number of processes running on a single node. If you plan to run two processes on node 0 and one process on all other nodes, the number entered should be two.

WARNING

If you are programming in Fortran, and you need to change the value for the maximum number of processes, you MUST change the value in the PRASE_FOR.H file as well as changing it in the prase_user.h file. This is somewhat cumbersome but must be done. The two values in the two different files must be the same. Each value is used to set up an array. If they differ, funny things can happen (of which were actually experienced during testing). If the value entered for the maximum number of processes (*pids*) is to small, errors are likely to be encountered at compilation time.

WARNING

If you enter a value that is too small for the maximum number of message types, the data will contain a value in one of the fields pointing this problem out. However, the translator will not flag an error and may cause the results to be viewed using Seecube to be erroneous. Make sure that this number is big enough to cover all your message types. If a 0 is entered for this value, an error is likely to occur at either compile time or link time. The error will refer to unresolved externals.

2.6 The PRASE_FOR.H File

There are two values listed in the PRASE_FOR.H file but only one should be changed. Please read the explanation given for the prase_user.h file and heed the warning. If you change the maximum number of pids value in this file, you MUST also change it in the prase_user.h file.

WARNING

If the value entered for the maximum number of processes (*pids*) is to small, errors are likely to be encountered at compilation time.

2.7 Example Flow

The following is an example of the commands on each machine you might use in a real session. The operating system prompts are contrived to make it clear as to what machine you are executing them on. Also, it is assumed that the Microvax and Sun Workstation share the same directories and data files. Details included in subsequent chapters of this manual are not included here.

```
vax% pvax
vax% mkdir newdirectory
vax% cd newdirectory
vax% prase_new
vax% vi prase.cfg (edit the configuration file)
  .
  .
  .

cube% pset
cube% mkdir newcubedir
cube% cd newcubedir
cube% prase_newcube
cube% vi prase_user.h  (edit the user include file)
cube% vi PRASE_FOR.H (edit this if a Fortran user
                      and changed the prase_user.h
                      file pertaining to max pids)
  .
  .
  .
```

```
sun% psun
sun% cd newdirectory (since already setup on vax)
.
.
.
```

If you have already instrumented a piece of code, and then make changes to it and try again, you will not need to execute the new commands or change the configuration or user include files. If, however, you desire to make changes to either of these files, you may then change them to reflect the new situation.

III. Preprocessing

Once setup is accomplished, you are ready to start using PRASE. The first part of PRASE for you to exercise, involves the preprocessing facilities provided. Two preprocessors have been provided; one for programs written in C, and one for programs written in Ryan-McFarland Fortran. These preprocessors require slightly different procedures so they will be discussed individually.

The preprocessors may not work correctly if the code provided to them does not compile without errors. If bad code is given to the preprocessors, results will be unpredictable. You should run every subroutine that has anything to do with the application you are attempting to monitor through the preprocessor. If you have separate files for separate subroutines, each file should be preprocessed. The preprocessors will take your normal Hypercube calls, and adjust the code so that the PRASE collection routines will be called in place of the normal Hypercube routines (the normal Hypercube routines are called by the PRASE collection routines). Only those calls stipulated in the configuration file will be effected.

3.1 C Preprocessing

There are basically two things you need to do to preprocess a C program. First, you must add a special comment in the main routine (subroutines should NOT contain this comment). Second, you must set up the configuration file to reflect your code and your monitoring requirements. Once these two items are accomplished, you can run the preprocessor to get code that will collect data.

The special comment you must add goes in a specific place in the main routine. This comment is to aid the preprocessor in getting oriented. The comment you should add is

```
/*#$*PRASE BEGIN MAIN*/
```

The comment MUST be added EXACTLY as you see it here (except that it may have leading and trailing blanks as desired). If it is not, preprocessing will not be accomplished correctly.

The comment needs to go after all local declarations in the main procedure but before any executable code. It should be on a line by itself. The swirly braces ({}) should be only one deep at this point. This means that only the left brace that opened the main procedure should be active when the comment is added. An example follows:

```
main()  {

int       x, y, z;
float     q, r;
char      mychar[25];

     /*#$*PRASE BEGIN MAIN*/

   x = 5;
   y = 10;
   z = 15;

   q = (float) x * y * z;

   }
```

The preprocessor will see this comment and place in the file some initialization code required by PRASE. It will also begin looking for the end of main so it can place a call to an ending procedure required by PRASE.

Once your code is ready, and you have set up the configuration file as previously discussed, you are ready to preprocess your file. Whether you are on the Microvax or the Sun, the command is the same. The command you will use was prepared by the set up script file you should have run when you logged on. The command you should now issue is:

```
% prase_c_pre
```

The preprocessor should tell you what it is doing as it does it. The input file name stipulated in the configuration file should end with a *.c* extension and be no longer than 12 characters including the extension. The preprocessor output file will have a *.p.c* extension in its place. This will contain the preprocessed code. Scan the new code to get a feel for what PRASE is doing with your code.

If you have problems, first make sure you have code that compiles. If you feel that you have found a legitimate problem, please report it. It is very difficult to consider every possible combination of valid code statements.

3.2 Fortran Preprocessing

Unlike the C preprocessor, the Fortran preprocessor does not require a user to enter a special comment. All you need to do is set up your configuration file and go.

There are some known situations that the preprocessor will not support. One such item is in reference to statement function statements as outlined in the Ryan-McFarland Fortran manual. If you use this Fortran feature in the main program of your Fortran code, the preprocessor will place executable code prior to the statement (which is incorrect syntax). This is an unsupported feature at this time. If you wish to edit the preprocessed code, however, you may fix the flow of statements by moving your statement function statement prior to the added PRASE code. This only applys to the main routine. Statement function statements in subroutines are not a problem.

Also, comment lines found in the middle of a call that is continued are not supported. For example:

```
CALL SYSLOG(MYPID(),

C  Comment in between start and end of a call that is continued

     *                 mymsgbuf)
```

Another unsupported feature pertains to logical if statements (this is a statement with no **THEN** part). If you have a logical if statement, with the statement portion being something you want to instrument, the preprocessor will not set up the instrumentation code. Other logical if constructs (not containing a call you want to instrument) should be fine. An example follows:

```
IF (X .GT. 77) CALL SYSLOG(MYPID(), MYMSGBUF)
```

A logical if statement with a **STOP** after the if is supported. If you have a situation as in the example, you should change it to an *if-then* type construct so that the preprocessor will correctly handle the call that you wish to instrument. Another problem, that also pertains to if statements, can be seen in the following example:

```
IF (STATUS(CI) .EQ. 1) GOTO 20
```

The problem lies with the call to *status* that is embedded in the if statement. Although you may have set up the configuration file so that this should get instrumented, it will not. You should call *status* prior to the if statment, placing the returned value in a variable, and then test the variable in the if statement.

Now, on to using the Fortran preprocessor. To execute the program, enter the following command:

```
% prase_for_pre
```

The program should keep you abreast of its actions. Once complete, the output file will be in a file with a *.p.f* extension (the original file name should be no longer than 12 characters including the extension). The original extension should have been of the form *.f*. Peruse the code to get a feel for how it is different.

Again, if you feel you have legitimate code, but the preprocessor seems to be messing you up; report the problem. With your help, the system can get better.

Once you have preprocessed your code, copy it to the Hypercube of your choice in preparation for actually collecting the data.

3.3 Example Flow

The following is an example of the commands on each machine you might use in a real session. The operating system prompts are contrived to make it clear as to what machine you are executing them on. Also, it is assumed that the Microvax and Sun Workstation share the same directories and data files. Details included in the set up chapter have been included here. Details from subsequent chapters are not included.

```
vax% pvax
vax% mkdir newdirectory
vax% cd newdirectory
vax% prase_new
vax% vi prase.cfg (edit the configuration file)
vax% cp ../oldcode . (get the old code into the
                      new directory)
vax% vi oldcode.c (if C only to add special comment
                   to the main routine only)
vax% prase_c_pre (or prase_for_pre for Fortran)
```

```
vax% ftp mem_cube (You can use rcp if you like,
                   you just need to now get your
                   code to a Hypercube)
  .
  .
  .
cube% pset
cube% mkdir newcubedir
cube% cd newcubedir
cube% prase_newcube
cube% vi prase_user.h (edit the user include file)
cube% vi PRASE_FOR.H (edit this if a Fortran user
                      and changed the prase_user.h
                      file)
  .
  .
  .
sun% psun
sun% cd newdirectory (since already setup on Vax)
  .
  .
```

The above includes all steps for code that was not previously instrumented. If you have already instrumented a piece of code, and then make changes to it and try again; you will not need to execute the new commands or change the configuration or user include files.

3.4 Things You Should Know

The preprocessor for Fortran allows access to the PRASE subroutines by providing calls to the C collection routines. Several variables are placed in the header of the main routine and the subroutines. One problem you might run into is if you are also trying to call C routines. The preprocessor will place new declarations into your file for CALLC, CSEG, etc. This means that if you have already defined them, you will get an error at compilation time. To alleviate this problem, simply edit the preprocessed file with the problem and delete one of the instances of the declaration only in the routine that has the problem.

If you experience any errors such as unresolved externals or errors and warnings, you should first look at the prase_user.h and PRASE_FOR.H file. Make sure that the numbers are correct in these files (for more information on these two files, see the discussion in this manual for each file). Also, you should make sure you are linking with everything that you need.

Another item you must handle is in reference to gathering data pertaining to *recv* calls (not *recvw*). When a *recv* is issued, the data in the return parameters is not always valid. You MUST also instrument *status* calls when you instrument *recv* calls. There is only one exception to this, if you never call a *recv* until you know data is available (this can be done using *probe*), then you do not need to gather data on *status* calls. However, if *probe* is not used in this way, you must gather data on *status* calls and actually call *status* in your program until it tells you that the data is valid. You may NOT issue another *recv* for the same channel or message type until the previous one is clear (you may issue a *recv* for a different channel or message type but must then call *status* for this *recv* as well).

You must do all this so that the PRASE data collection routines record the correct data. If this is not done, then the data will be incorrect and could lead to erroneous results being given to you.

IV. Data Collection

To collect data pertaining to your program, you should have already preprocessed your code. Once on the Hypercube, you should set up a new directory and start out with just your code being present. Since you are planning to use PRASE, you should have already executed the **pset** command which sets up certain aliases for your use.

Once you are in your new directory with the preprocessed code file, you need to copy certain files into your area. To do this (for a new piece of code you want to monitor), execute the following command:

```
% prase_newcube
```

This will copy several files into your directory as well as create a subdirectory for storing the collected data. You now have your own copy of the PRASE data collection subroutines. This allows you, as the user, to tailor PRASE to whatever configuration you like.

You now should edit the prase_user.h file and change it as discussed previously. If you are using Fortran code, you should also edit the PRASE_FOR.H file to make sure that the maximum *pids* parameter agrees with what is in the prase_user.h file. IT IS VERY IMPORTANT THAT THESE TWO FILES AGREE.

Once these two files are in order, you are ready to compile your own copy of the PRASE collection subroutines. To do this, you should execute the following command:

```
% pcode_compile
```

Now you should compile your code. If you are compiling C source routines, you should compile using the Huge (-Alhu) switch and the -K switch (see the XENIX

286 Programmer's Guide Chapter 2 - this switch "Removes stack probes from a program" which are "used to detect stack overflow on entry to program routines"). The Huge switch adds some extra abilities for the code to cross 64K byte boundaries. This is important for the subroutines to work properly.

CAUTION

If you do not use the -K option, you may get an error at load (link) time referring to a procedure that apparently has something to do with the stack.

If you are compiling Fortran code, you should use the -bhkze options and the -w option. The -w option suppresses warning messages. If you compile a preprocessed file without the -w option, several warnings will appear because of the C calling routines. These may be ignored as long as they pertain to different types of arguments or different numbers of arguments for the calls to C. Therefore, unless you need to see warnings, it is much cleaner to use the -w option.

Next comes linking. You more than likely already have a makefile with certain libraries being linked. Here is a list of libraries you should link with. You may, however, experiment with these by deleting one at a time to see if in fact they are required.

For C you should try

```
/lib/Lseg.o
/usr/ipsc/lib/Lcrtn0.o
prase code.o   (definitely needed)
your stuff
/usr/ipsc/lib/Llibcnode.a
```

For Fortran you should try

```
/lib/Lseg.o
/usr/ipsc/lib/Lcrtn0.o
prase code.o   (definitely needed)
your stuff
/usr/ipsc/lib/Llibcnode.a
/usr/ipsc/lib/Llibfnode.a
```

Once the link is done you are almost ready to go. Next, you should get the cube for use by using the **getcube** command. The very next command should be a **cubelog** command so that log messages will go to your personal file. This is important because PRASE may write an error message to syslog that you should see. After this, you should load the cube for the appropriate dimensions.

PRASE dumps data to a program that runs in the background on the host. This is why data can be dumped at any time during a run. The host process sits waiting for information from the collection subroutines and when data does arrive, writes it to disk. You must start this routine AFTER the **load** (this is the **load** for loading the appropriate dimensions and resetting the Hypercube nodes, not for loading a user's code) has been accomplished. You MUST start this while your default directory is the new directory that you created. The routine will attempt to write its data to a subdirectory call prase_data (this was built for you by the **prase_newcube** command).

To start this program you should execute the following command:

```
% prase_clct
```

At this point you are now free to run your code by either loading it directly or starting a host process that loads it for you. Once the run is complete (you should

watch the lights or give PRASE some time to finish dumping the data that was collected), you should have data files in the prase_data directory. There should be one file for each process that ran. The file names have the format n#_p$ where # stands for the node number and the $ stands for the process id (*pid*) of the process that sent the data to the host. The data is in binary form. You may, however, view this data by using a command set up for you called **pv**. To look at the data simply type **pv** and the directory and file name. For example, the command

```
% pv prase_data/n0_p0 | more
```

would display the data collected on node 0 *pid* 0 and pipe the output through **more** so you can look at it a page at a time. If you wish to get an ascii copy of this data, simply type the following

```
% pv prase_data/n0_p0 > x
```

with the ascii data being placed in file x.

If you wish to automatically manipulate this data (for instance translating and using Seecube), you must get it back to the Sun/Microvax environment. To do this you can use either rcp or ftp. If you use ftp, you MUST BE SURE TO PUT FTP INTO BINARY MODE PRIOR TO PASSING THE FILES. If you do not, erroneous results are likely. To put ftp into binary mode, simple type **binary** once you have an ftp prompt.

Before you release the cube, you should stop the host collection routine. You should execute a **ps -a**, look for the *pid* of the prase_clct program, and then issue a **kill -9** *pid* command. This will stop the collection routine. You may then release the cube.

Before you log out, you should remember to check the log file you set up with **cubelog**. Look for any errors initiated by PRASE. There are two error messages

you may see that are written by PRASE and a third that is provided by the system. The first is:

```
PRASE NOTE: Ending with an outstanding recv
```

The error message means that a *recv* was issued but PRASE did not recognize a completion for the *recv*. This could happen for a couple of reasons. First, your program may continue to issue *recv* services not knowing when program completion occurs. In this case, if one *recv* was left open, all might be just fine. In another case, a *recv* may have completed but PRASE did not pick up the completion. This could happen if you neglected to call *status* after every *recv* was issued. Remember that you must continue to call the *status* service for the open *recv* until the system provides a status that the message has been received. You should determine which case you are in. The former is a fine condition that should not cause a problem in analyzing your data. The latter, however, means that the data could be corrupted because one *recv* somewhere did not register the appropriate ending. Once you issue a *recv* for a particular message type or channel, you should not issue another until the one issued clears (you can know when the message clears by calling *status* until the appropriate value is returned).

The second error message you may see is:

```
PRASE ERROR: Too many outstanding recv calls
```

This message means that you have exceeded the maximum number of outstanding *recv* calls that you can have at one time. The current maximum is 20. The impact of this message is that the data can no longer be trusted as valid.

The system error message will be something like the following:

```
Warning: Spilling data segment from LBX to Node board
Warning: Spilling data segment from LBX to Node board
Warning: Spilling data segment from LBX to Node board
Warning: Spilling data segment from LBX to Node board
Loader: Out of memory on node and LBX boards.
```

This means that you are trying to use too much memory. This could happen for different reasons. You may have large data structures and by adding the instrumentation code have exceeded available space. You may have asked to save too many records in the trace (as setup in the prase_user.h file). Each data record takes 32 bytes. To save space, lower the PRASE_DUMP_THRESHOLD value. This will save you 32 bytes for each integer value that you lower it.

If at some point you reload the cube or issue a **loadkill** command, and the collection routine is running, it will abort with a message something like

```
call: recvmsg pid: 32767 error: Channel went bad
```

Simply restart the collection routine (if needed) prior to reloading your code onto the Hypercube.

4.1 Example Flow

The following is an example of the commands on the Hypercube that you might see in a real session.

```
% pset
% mkdir newcubedir
% cd newcubedir
% prase_newcube
% vi prase_user.h
```

```
% vi PRASE_FOR.H (for Fortran users who changed
                  prase_user.h)
% pcode_compile
% rmfort -bhkze -w   or   cc -Alhu -K (compile your code)
% vi makefile (add any necessary object or library
               modules to the make)
% make mycode (link the code to produce an executable)
% getcube
% cubelog -l mylog
% load -c 5  (load the cube with some appropriate
              command)
% prase_clct
% somehow run your program
% pv prase_data/n5_p0 (view the data in ascii for this
                       particular node and pid)
% more mylog (look for PRASE errors)
% ftp sunworkstation (use binary - just type it in at
                      the ftp prompt) (send your data
                      files back)
% ps -a (look for pid of prase_clct)
% kill -9 pid
% relcube
```

V. Translation

Now that you have collected data, you may want to look at that data using the graphical routines provided by Seecube. To do this, you must first translate the PRASE data into Seecube data. Only data that was collected in an environment where single processes were running on each node can be translated and used. To use the translation program, you must either be on a Sun or a Microvax that supports PRASE.

In preparation, you must have the configuration file you used for preprocessing, and the data files collected on the Hypercube. You should be in the directory originally made for the PRASE session (because eventually there are some files Seecube will need).

Once you are in the right place with the right stuff, enter the following command

```
% prase_trans
```

The translator should tell you what it is doing. The ultimate output is a file call human which is used by Seecube. All but the necessary records are discarded in the translation phase to the human file. The translator is really only interested in message passing data records. The non-applicable records are filtered out.

The translator correlates sends from one process to receives on another process. Essentially, it tries to match a send from one process to the corresponding receive. If it cannot do this, it will tell you at the end of the run how many records were not correlated.

This could happen for several reasons. First, if you start data collection at a time other than time 0 (as set forth in the configuration file), a send may not be recorded whereas the corresponding receive might. Thus, we have a situation where

we will not correlate records. A second situation would occur if you used up your quota of records prior to the end of a run. This means one end of the send-receive might get recorded while the other might not. The third situation in which this could occur is if the data is bad. If you suspect bad data, try using the **pv** *command* on the Hypercubes to look at what you got.

VI. Using Seecube

To understand the workings of Seecube better, you should read the Seecube User's Manual. Here we will present what you will need to get the data in the right format and to get the graphical displays going. To use the Seecube software, you must be on a Sun workstation.

Once translation has occurred, and you have a file called human; you are ready to use the Seecube software. Prior to the graphical displays, however, you must execute the Seecube Resolver. This command has been set up for you by the setup procedure. To execute the Resolver, do the following

```
% resolve
```

You will get several lines of printout from the program. Ultimately, you should have some files called interlaced (some with file extensions). Previously brought in are also some files that Seecube will use. They are the context file, and the files found in the colors and clumps directories.

CAUTION

It appears that the file called *human* can get too big to be handled by the resolver. You should be aware of this possibility as you run the Resolver.

To use the graphical displays, simply enter the following

```
% seecube
```

You will be presented with the Seecube control panel. From here, you should refer to the Seecube User's Manual for specifics pertaining to the possible displays.

If you have redefined the delete key in your login file, you may have trouble changing parameters in Seecube. This is important if you desire to change certain values for some of the displays.

CAUTION

Seecube allows a user to specify how many milliseconds are advanced every time the displays are updated. If this value is set to anything other than one (which is highly likely and desirable), when the clock gets near the end of the data, it will not complete the run unless it can advance to exactly the correct value for the last millisecond in the run. You need to be aware of this when looking at the end of a run, because you might think (as the author did once) that messages had not been handled at the end of the run. Just set the slider for this value to one at the end of the run so that the displays can complete. Messages could also appear to be left on queue at the end of a run if PRASE did not have enough room to collect all the data for an entire run.

CAUTION

Occasionally, because of the times recorded in the trace, it may appear that a message is sent in zero or negative time. It appears to be the case that if the times reflect this situation, the Seecube displays will not graphically show the presence of the message.

VII. Collecting Your Own Data

PRASE provides utilities that allow the user to collect data of his own that is autonomous from the Hypercube calls. You can store different variables of differing data types into the PRASE data trace. This will hopefully give you some idea of how certain variables are changing over time and how they compare across processes and nodes.

To actually collect this data, you must place the calls into your code by hand. No automatic placement is supported at this time. The next two sections give the syntax you will need to introduce these calls into your program. Once you have added the calls, you should then run your code through the appropriate preprocessor to complete the process of preparing to collect data.

7.1 C Support and Syntax

Following the example in the Intel manuals, the calls to each of the routines will be provided with any declarations that are pertinent.

- Integer Variables

```
int       myvar;

PRASEMARKINT(&myvar);
```

- Long Variables

```
long      mylongvar;

PRASEMARKLONG(&mylongvar);
```

- Short Variables

```
short       myshortvar;

PRASEMARKSHORT(&myshortvar);
```

- Float Variables

```
float       myfloatvar;

PRASEMARKFLOAT(&myfloatvar);
```

- Double Variables

```
Double       mydoublevar;

PRASEMARKDOUBLE(&mydoublevar);
```

- Character Variables

```
char       mycharvar[13];

PRASEMARKCHAR(mycharvar);
```

Although you can send a very large character string to the character routine, only the first 13 characters will be saved. Be aware that if you have bad data in part of the variable, that it may get written to the trace.

CAUTION

Do not pass the PRASEMARKCHAR a variable that is one character in length (this would mean that the variable was not a pointer and thus might cause erroneous results).

7.2 Fortran Support and Syntax

Following the example in the Intel manuals, the calls to each of the routines will be provided with any declarations that are pertinent.

- Integer*2 Variables

```
      INTEGER*2     INT2VAR, RESULT
      EXTERNAL      PRASEMARKINT

      RESULT = CALLC(PRASEMARKINT, 2,
     *                COFF(INT2VAR), CSEG(INT2VAR))
```

- Integer*4 Variables

```
      INTEGER*2           RESULT
      INTEGER*4           INT4VAR
      EXTERNAL            PRASEMARKLONG

      RESULT = CALLC(PRASEMARKLONG, 2,
     *                COFF(INT4VAR), CSEG(INT4VAR))
```

- Real*4 Variables

```
      INTEGER*2         RESULT
      REAL*4            R4VAR
      EXTERNAL          PRASEMARKFLOAT

      RESULT = CALLC(PRASEMARKFLOAT, 2,
     *                COFF(R4VAR), CSEG(R4VAR))
```

- Real*8 Variables

```
      INTEGER*2        RESULT
      REAL*8           R8VAR
      EXTERNAL         PRASEMARKDOUBLE

      RESULT = CALLC(PRASEMARKDOUBLE, 2,
     *                COFF(R8VAR), CSEG(R8VAR))
```

- Character Variables

```
      INTEGER*2        RESULT
      CHARACTER*13     CHVAR
      EXTERNAL         PRASEMARKCHAR

      RESULT = CALLC(PRASEMARKCHAR, 2,
     *                CCHOFF(CHVAR), CCHSEG(CHVAR))
```

Notice the difference in the character calling sequence. Instead of COFF and CSEG, CCHOFF and CCHSEG are used. Although you can send a very large character string to the routine, only the first 13 characters will be saved.

CAUTION

Do not pass the PRASEMARKCHAR a variable that is one character in length (data may be erroneous). You may pass a single character in a character variable with more than one character present. Be aware that if you have bad data in part of the variable that that data may be written to the trace.

WARNING

If you neglect to add the EXTERNAL statement as shown in the examples above, you are likely to get exception errors when you attempt to run your code.

VIII. Very Important! Please Read

There are certain things that as a user of PRASE should know before using the software. This chapter will list these items. Please read them all, it could save you some grief and erroneous results. The items are not in any specific order, they are just here for your information.

- Number of Channels - PRASE opens and uses a channel to communicate with the host collection routine. This means that there will be one less channel available for your general use.
- Reserved Message Types - PRASE uses two message types for specific purposes. You should NOT use these two types. They are 32766 and 32767.
- Reserved Pid for copen - The collection routine that runs in the background on the host opens a channel using the pid number 32767. Then, when processes wish to communicate with it, they simply send to that pid. You MUST not use this pid number.
- Configuration File Correctness - Make sure that you include all required groups correctly (as outlined in the example configuration file provided). If required items are left out, unusual things can happen causing errors or erroneous results to occur.
- Message Loss Assumption - When translation is accomplished, correlation is done based on a count in the record. The count gets updated as messages come into a node. Separate counts are kept for each different message type coming from a specific node and *pid*. The counts are kept on each node and should correspond between send and receive records for a specific message. If a message was lost, these counts would get out of phase and would cause records to correspond incorrectly. A basic assumption has been made that the Hypercube will not lose a message once it has been given to the system.

- Speed Degradation - Counters are kept for each message type that you use. The more message types that you have, the longer it takes PRASE to find and update the counter. Thus, with more message types, you can expect more degradation in speed.
- Multiple Receives - If you plan to issue multiple receives without wait (*recv*) at the same time, you MUST use different variable names for the returned values and you MUST call *status* for each until a good status is returned. Remember, you should not have multiple receives outstanding for the same channel or message type.
- Fortran Columns 73 to 80 - In some cases, the data in these columns may not show up in the preprocessed code. Since this data is not executable, this should not cause an error.
- Filename lengths - Original file names entered in the configuration file should be no longer than 12 characters including the extension. This leaves room for the preprocessor to add two characters to the name and still have the name be correct on the Hypercube.
- Host procedures loading code on the Hypercubes - You may want to load your node programs on the Hypercubes using a host program rather than loading them directly to the nodes. The name of the file to load to the nodes might be compiled in the code. There are several ways to handle this problem. You can move the executable to the old name, or change your code to reflect the change.
- Killing the node processes - An example program provided by Intel was found to stop the node processes by issuing a **loadkill** from the host rather than having them run to completion. In this case, some or all of the data collected may not get dumped. If you are doing this be aware of the situation. You can lower the dump threshold so that some data will be received or you can modify your program to stop in a different way.

- Using *_exit* - If you are using *_exit* in C, when instrumentation occurs, this will get changed to a prase_exit routine. This is so that we can end PRASE correctly prior to exiting. This is also handled with a call to *exit*. However, the replacement routine for *_exit* does not call *_exit* after the PRASE ending is complete. It instead calls *exit*. *exit* and *_exit* have different functions.

- Begin and End Times for *send* - When a *send* (not *sendw*) service is instrumented, the times recorded mean something different than what you might expect or what might be shown on Seecube. For a *send*, a begin time and an end time get collected. The end time is the time after the *send* call completes, NOT the time the buffer becomes free. When using Seecube, it might appear that a message is all written when in fact it was still in the process of writing. When using Seecube, the display should be interpreted in light of the way the PRASE data was collected (if in fact you have instrumented the *send* service).

- *recv* Problem - When a *recv* is issued, if a message is not ready for receipt, then the data instrumentation record does not get recorded at that time. However, the begin time does get saved internally. If you call *recv*, and find that it has not completed, you MUST not call *send, sendw, recvw, copen,* or *cclose* prior to the completion of this event. If you do, then the trace shows times out of order. This is only a problem if you plan to use Seecube. If you decide to do you own data analysis, you must realize that the time ordering is not necessarily maintained when calling *recv*. This means that the begin time of one record may be later than the begin time of the next record.

  You may call another *recv* (only on a different channel with a different message type and different variable names), however you should not call *status* for the second *recv* issued until the first *recv* issued clears.

  You can detect this problem by scanning the output from the **pv** utility on the Hypercube. If a time is out of order in a trace, then the results you will see on Seecube will probably be erroneous. If however, you are planning to analyze

the data separate from Seecube, then you should proceed with your analysis with the knowledge that times may be out of order. Currently, there is no fix other than the guidelines given above and there is no known way to identify the problem other than by using the **pv** utility.

- Count Field Errors - As you use the **pv** command, you may find the count field containing negative numbers. These numbers mean certain things. When the maximum number of message types is smaller than the number of types a user attempts to use, there is a problem. The only indication of the error is in the trace itself. A value of -99 is placed in the trace in the count field. This was by design. However, if a user does not examine the trace, and just processes the data, he is likely to get erroneous results. At some point the *syslog* should contain an error message for this. However, at this time, you should make sure that the number you enter in the prase_user.h file is big enough to encompass all possible message types.

  A -999 or -9999 means that PRASE probably has some internal problems. If one of these counts are received, the data is most likely invalid and should not be used for analysis.

- Dangerous *cc* Option - There is an option listed in the XENIX(XENIX is a trademark of Microsoft Corporation) 286 Programmer's Guide for *cc* that "Reverses the word order for long types (this was found on page B-23 of that guide)." Although never attempted, this option will probably render the data files useless if translation is attempted. This option should be used with extreme caution if at all. The option is *-Mb*.

- Instantaneous Events - Occasionally, because of the times recorded in the trace, it may appear that a message is sent in zero or negative time. It appears to be the case that if the times reflect this situation, the Seecube displays will not graphically show the presence of the message.

- Global Sends - The Hypercube supports a global send capability where a user can issue one *send* (or *sendw*) to send a message to a subcube or the entire Hypercube. PRASE does not support this function. Using this function while also using PRASE may cause data to be erroneous. You should not attempt this.

- Configuration File Updates - If you have an existing piece of code that has been instrumented, and you change the configuration file, you must begin the PRASE process again at the preprocessing step.

- Instrumenting Fortran calls to *syslog* - If you have something like the following:

  CALL SYSLOG(mypid(),'my msg')

  The message portion will show up in the syslog with all blanks removed and all letters being capitalized. This is because the Fortran preprocessor changes things around so that a variable can be passed instead of a character group such as illustrated.

- Errors - If you experience any errors such as unresolved externals or errors and warnings, you should first look at the prase_user.h and PRASE_FOR.H file. Also, you should make sure you are linking with everything that you need.

Appendix A. *Configuration File Example*

The following is an example of a configuration file that could actually be used. There are lots of comments provided to aid in properly setting up your own configuration file. When you do the initial setup to use PRASE, a copy (without all of these comments) of a configuration file will be placed in your default directory. Following the guidelines given here, should help you in getting your configuration file right.

Example: prase.cfg

```
# This is a sample configuration file used by the Parallel
# Resource Analysis Software Environment (PRASE).  The rules
# for building your own config file as well as examples are
# included.  Please read all of what is contained in this
# file prior to trying any modifications.
#
#   This configuration file was written by Capt Mark Kahl
#
#
#  The prase.cfg file is used by three programs within the
#  PRASE system.  The preprocessors use the information in
#  this file to build a preprocessed version of a user's code.
#  This file tells the preprocessors what nodes will be
#  running code for this run, how many processes (and their
#  pid numbers) will be running on each node, what calls the
#  user wishes to instrument, the file names to preprocess,
#  and finally, which libraries or object modules are needed
#  to link with other than the defaults provided by the
#  preprocessor (the libraries part is not currently
#  implemented). Another program, which translates PRASE
#  collected data into Seecube (copyrighted by Alva Couch
#  at Tufts University in Massachusetts) format,
#  uses this file only for the node and pid information.
#
#  It is IMPORTANT to use the same configuration file
#  throughout the preprocessing, data collection, and
#  translation phases of a PRASE session.
#
#
#
# THE FOLLOWING PROVIDES INFORMATION PERTAINING TO SPECIFIC
```

```
# FILE FORMAT. PLEASE FOLLOW THE RULES EXPLICITLY.
#
#
# -------->   COMMENTS AND BLANK LINES:
#
# This is a comment line since it starts with a pound sign.
# Comments may only fit onto one line.  If you want to say
# more, go to the next line.  DO NOT let a comment line wrap
# by itself.
#
# WARNING:  You can leave blank lines but they must have
#           nothing on them.  This means not even blanks
#           can be on the lines.  Suggestion: Don't use
#           blank lines.
#
# DO NOT ALLOW ANY TYPE OF LINE TO WRAP AROUND.
#
#
#
# -------->  GROUPS:
#
# Each configuration file should contain what are called
# "groups" of information.  Each group has a specific
# purpose and MUST be formatted by following certain rules.
# Examples of each group are provided below.  Each of these
# groups MUST be present (except for the LIBRARY group
# which is currently not used) prior to beginning a PRASE
# session so read the instructions and make up your own file.
#
#
#
# -------->   NODE GROUP:
#
# The following deals with NODE groups.  Its purpose is to
# convey to the PRASE programs which nodes are running code.
# The information can be put together into "like" groups,
# or listed node by node.
# Here are the format rules:
#
#   1.  Begin the group by placing the word NODE on a single
#       line starting the word in column 1 and insuring that
#       all capital letters are used.  (Remember: follow the
```

```
#       directions exactly)
#   2.  Nodes can be clumped in groups if each node in the
#       group is running the same number of processes with
#       the same pid numbers.  For example:  If nodes 0
#       through 4 are running one process each with the same
#       pid number of 7, then these can be placed in one
#       group.  However, if nodes, 1, 3, 5, and 7 are alike
#       but 2, 4, and 6 are different from the odd nodes,
#       then no grouping can be done.  What we are saying
#       is that to group like nodes, they must be in order.
#   3.  So, in step 1 we put the word NODE in columns 1
#       through 4; now, on the next line, we place the
#       beginning node number for this group of nodes.
#   4.  On the next line we put the ending node number
#       for this group.  This number can be the same as the
#       beginning node number (saying that this group
#       pertains to only one node), but cannot be smaller
#       than the beginning node number.
#   5.  Only list a single node once.  If you list it more
#       than once, only the last listing will be accepted.
#   6.  Once a node group is completed, a PID group MUST
#       follow immediately.
#
# Now, here is an example of a node group.  We will leave
# the comment in front of the group.  If you really wanted
# this grouping, all you would need to do is delete the
# comment marking (insuring that the name NODE began in
# column 1).  This example says that this group pertains
# to nodes 0 through 10 or eleven nodes total.
#
#NODE
#   0   begin node
#   10  end node
#
#
# -------->   PID GROUP:
#
# Next let's give rules and an example for a PID group.
# First, the rules.
#
#   1.  The first thing you must do for a PID group is
#       insure that there is a NODE group as the group
```

```
#       immediately prior to the PID group.  The PID
#       group goes hand in hand with the NODE group.
#       The PIDs listed in this group are the pids that
#       will go along with the NODE group.  Thus, the
#       PIDs listed below will be valid for the NODES
#       listed above.
#   2.  Now, put the PID group identifier on the next
#       line and make sure it is in capital letters and
#       starts in column 1.
#   3.  On the next line, you should put the number of
#       processes that will run on each individual node
#       (each process must have its own unique pid, thus
#       we say pids almost synonymously with process).
#   4.  If you wanted to run three separate processes on
#       the nodes listed in the previous node group, you
#       would put a 3 in the position for number of pids.
#       This number must be at least 1 and less than or
#       equal to 20.
#   5.  The next line(s) contain the actual pids that
#       you plan to run on the nodes.  If you have ten or
#       less pids running on a node, then all the pids
#       should be placed on one line with blanks
#       separating them.
#   6.  If you plan to run more than 10 pids on a node,
#       the first 10 must go on the first line and the
#       remainder of the pids should go on a second line.
#
# EXAMPLE:
#
#NODE
#  0   begin node
#  10  end node
#PID
#  12  number of pids to run on the nodes above
#  0 10 20 35 44 127 329 451 77 1054  These are the first 10
#  88 12453                           These are the last 2
#
#
# The above example has nodes 0 through 10 running 12
# processes each with the twelve pid numbers being the
# pids of the processes running on each node.
#
```

```
# NOTE: IF YOU PLAN TO RUN THE DATA YOU COLLECT THROUGH
#       THE SEECUBE SOFTWARE, THEN YOU SHOULD ONLY BE
#       RUNNING 1 PID PER NODE.
#
#
# The following is the actual node pid configuration
# for this run.
#
NODE
    0
    31
PID
    1
    0
#
#
#
# ------->   INSTRUMENT GROUP:
#
# The next section we will discuss pertains to the
# calls that the user would like to instrument.  You
# may want to gather data just on calls to sendw and
# recvw hypercube calls.  Or, you may want to
# determine how often you call the greenled service
# so you may just want to collect data on that.  Or,
# you may want to collect data on everything there
# is to collect data on (or even some subset).  This
# section deals with telling the preprocessor which
# items you want to collect data on.
#
# Before getting into format and specific rules,
# let's give a list of all the calls that you can
# instrument.  The following list is in alphabetical
# order.
#
#   CCLOSE, COPEN, FLICK, GREENLED, PROBE, RECV,
#   RECVW, REDLED, SEND, SENDW, STATUS, SYSLOG
#
# The above calls are the only hypercube calls that
# can be instrumented.  What this all means is that
# the users original calls to the routines chosen
# will be replaced with calls to data gathering
```

```
# routines that collect data about the actual
# hypercube call.  Other routines may be called to
# collect data but must be entered manually.
#
# Note that if you are instrumenting data to
# ultimately play back on the Seecube display
# software, you should instrument the COPEN and
# CCLOSE calls as well as sends and receives.
#
# Now we can describe the new group.  The group
# will contain group the name INSTRUMENT to start.
# With all this in mind, let's list the rules.
#
#   1.  The group is identified by placing the
#       word INSTRUMENT in column 1 of a line.
#       Make sure it is all in caps.
#   2.  The next line should contain a number
#       signifying the number of items you have
#       chosen to instrument.
#   3.  On subsequent lines, each of the calls
#       you wish to instrument should be listed,
#       one per line, in caps.  The names MUST
#       start in column 1.
#
# NOTE: When you say you want to instrument the
#       call to RECVW, then everywhere in the
#       instrumented piece of code that the call
#       to RECVW appears, it should be replaced
#       with a call to the data gathering software
#       (unless it is in an unsupported position
#       in the code - see the User's Manual for
#       details which will in turn call the RECVW
#       function for the user).
#
# EXAMPLE:  Let's say you want to gather data on all
#       message passing events.  Thus, you want to
#       know about every copen, cclose, send, sendw,
#       recv, recvw, and status (you should instrument
#       this if you instrument recv - there is one
#       exception, to this so see the User's Manual).
#       You also decide to look at how SYSLOG calls
#       affect the whole situation.  The following
```

```
#       example shows how to do this.
#
#INSTRUMENT
#   8  Number of calls to instrument
#COPEN
#CCLOSE
#SEND
#SENDW
#RECV
#RECVW
#STATUS
#SYSLOG
#
#
#
# With that in mind, here is where you should enter
# the actual commands for the preprocessor.
#
#
INSTRUMENT
    12  Number of calls to instrument
CCLOSE
COPEN
FLICK
GREENLED
PROBE
RECV
RECVW
REDLED
SEND
SENDW
STATUS
SYSLOG
#
#
#
#
# ------->   LIBRARY GROUP:
#
# THE LIBRARIES OPTION AND MAKEFILE OPTION ARE NOT
# CURRENTLY IMPLEMENTED!!!!
#
```

```
# Now we come to the libraries group.  This group
# denotes all the libraries and object modules
# besides the defaults that a user wants to link
# with.  This needs to be put in here because the
# preprocessor will build an appropriate makefile
# (not currently implemented) for the preprocessed
# code.
#
# Now for the rules.
#
#   1. Enter the group name LIBRARY on the first
#      line of the group and make sure it starts
#      in column 1 and is in capital letters.
#   2. On the next line, enter the number of
#      additional libraries/object modules you
#      wish to add.  The maximum is currently 20.
#   3. On subsequent lines add ONE (and only one
#      per line) entry per line.  Each line should
#      contain the full path name of the library
#      or object module.
#   4. If you do not need to add any additional
#      libraries or object modules, you may leave
#      this group out of your configuration file.
#
#
# Now for an example.
#
#LIBRARY
#   3
#/usr/eng/mkahl/objects/code.o
#/usr/ttt/user/x.a
#/usr/ttt/ttt/ttt.o
#
#
# The following is a real group.
#
#
LIBRARY
   3
/usr/eng/mkahl/thesis/code/sunada/x.o
/usr/eng/mkahl/thesis/code/sunada/lib.a
/usr/eng/mkahl/thesis/code/try.o
```

```
#
#
#
# ------->   STARTTIME GROUP:
#
# The following group tells the prase code when
# to start recording data.  The value must be
# given in milliseconds (no decimal points - just
# an integer).  The PRASE software will not start
# logging general data until on or after the time
# has been reached.  The time is a relative time
# from the beginning of the run.  Let's list the
# rules and then give an example.
#
#   1.  On the first line, starting in column 1,
#       place the group name STARTTIME (it must
#       be in caps).
#   2.  On the very next line, include the start
#       time in milliseconds that you want to
#       start recording data approximately relative
#       to the time that the code starts to run on
#       the Hypercube nodes.
#
#   Example:
#
#STARTTIME
#   1500
#
# The above example would cause recording to start
# approximately 1.5 seconds into the run.
#
#
STARTTIME
  700
#
#
#
#
# ------->   OLDFILENAME GROUP:
#
# This next group holds the input file name(s).
# This group is called the OLDFILENAME group and
```

```
# provides the preprocessor with the input name(s)
# of the code to preprocess.
#
# The rules are as follows.
#
#   1. Enter the name OLDFILENAME on the first line
#      of the group insuring that it is in CAPS and
#      starts in column 1.
#   2. On the next line, enter the number of files
#      to process.  This should equal the number of
#      filenames that follow.  The current maximum
#      is 250.
#   3. On subsequent lines, enter the filename(s) you
#      wish to use for input.  Make sure you start
#      the name(s) in column 1.
#   4. Each name MUST come on the very next line with
#      no comments or blank lines in between.  Each
#      name MUST start in column 1.  DO NOT
#      INCLUDE ANY TRAILING BLANKS!!!!!!!!!!!!
#   5. Names must not include a path, only files in
#      the local directory should be included.
#   6. Names MUST not exceed 12 characters including
#      the extension.  This is because the preprocessor
#      will add two characters to the output name and
#      the Hypercube will not accept more than 14
#      characters per name.  Fortran files should end
#      with a .f extension and C files should end
#      with a .c extension.  A name of test.f will be
#      output as test.p.f to show that it is a
#      preprocessed file.  Although this is something
#      you should adhere to, the preprocessor will
#      accept names up to a length of 33 characters
#      allowing you to preprocess long file names and
#      then rename the output files at file transfer time.
#   7.  A maximum of 250 files may be preprocessed at
#       once.  If you want to preprocess more files than
#       that, do it in several runs.
#   8.  You cannot preprocess C and Fortran files using
#       the same configuration file.  You must have
#       separate files.
#
#
```

```
# An example.
#
#OLDFILENAME
#myname.f
#myname2.f
#myname3.f
#
# Now for our real group.
#
#
OLDFILENAME
  11
for1.f
for2.f
for3.f
for4.f
for5.f
for6.f
for7.f
for8.f
for9.f
for10.f
for11.f
#
#
#
```

Appendix B. *User File Example*

The following is an example of an actual prase_user.h file that you can use. Comments are provided to aid in your understanding of what actually needs to be done to set up the user file.

Example: prase_user.h

```
/*  Description:  This file is meant to edited by the
    user to tailor the PRASE data collection software
    for his specific application.

    Four values are at the user's disposal.

    1.  PRASE_DUMP_THRESHOLD tells PRASE how many records
to save prior to dumping data to the host.  If
you want to debug and dump every record (so that
you don't lose data because of a crash), then set
this value to 1.  If you want to disturb your
application as little as possible (so that no
dumping occurs), set this value equal to the
value used for PRASE_TOTREC_THRESHOLD.
This value actually sets aside memory so you  must
consider the memory/speed tradeoff.  This value is
used to dimension the array that holds the records
in memory.

    2.  PRASE_MAX_MSG_TYPES tells PRASE how many different
message types your application will be using.  This
should be at least as big as the number of different
message types you use.  To protect yourself you should
probably set this greater than the number you have
just in case you make a mistake counting.  The
penalty you pay for setting this large is that you
give up memory (it is a definition for an array).

    3.  PRASE_MAX_PIDS tells PRASE how many different
processes you plan to run on a single node.  You as
a user should know this up front, so you should be
able to set this exactly.  Again, if you set it
higher, you lose memory space.
```

```
    4.  PRASE_TOTREC_THRESHOLD tells PRASE how many total
records to save for any particular run.  This can be
as big as you want as long as the disk will hold
all the data, however, voluminous amounts of data
may not be what you want.

*/



#define PRASE_DUMP_THRESHOLD 4000
#define PRASE_MAX_MSG_TYPES 30
#define PRASE_MAX_PIDS 1
#define PRASE_TOTREC_THRESHOLD 4000
```

Appendix C. *Format Breakout for* **pv**

The PRASE files that hold the information collected on the Hypercubes have their own unique format. This format can contain up to 12 fields. Depending on the type of record, the field meanings can vary. The number of fields in a record can also vary depending on the record type.

The fields discussed below correspond to the fields you will see when using the **pv** command.

The following explanations give each field and what they can mean depending on the record type. Find the field you are interested in, and then find the record type that you are interested in to determine the meaning for that field.

Record Format

- Field 1 (char 6) - This field can contain one of the following record types: *close, dump, end, flick, grled, init, mchar, mdble, mflot, mint, mlong, mshrt, open, probe, recv, recvw, rdled, send, sendw, stat, slog, wrapr,* or *wraps.* All other fields must be interpreted based on what type is entered in this field. The following list describes the meaning of each record type.

  - *close* - The record corresponds to a *cclose* event.
  - *dump* - The record corresponds to a dump of data. When you find this record type, it is telling you that PRASE has dumped data. When all data is kept in memory and then dumped at the end of a run, a dump record will not be included.
  - *end* - The record corresponds to the end of collection (which should also correspond loosely to the end of the run - unless the call to the ending routine was moved by the user).
  - *flick* - The record corresponds to a *flick* event.
  - *grled* - The record corresponds to a *greenled* event.

- *init* - The record corresponds to PRASE initialization.
- *mchar* - The record corresponds to user collected data of the *character* type.
- *mdble* - The record corresponds to user collected data of the *double* type for C and the *REAL*8* type for Fortran.
- *mflot* - The record corresponds to user collected data of the *float* type for C and the *REAL*4* type for Fortran.
- *mint* - The record corresponds to user collected data of the *int* type for C and the *INTEGER*2* type for Fortran.
- *mlong* - The record corresponds to user collected data of the *long* type for C and the *INTEGER*4* type for Fortran.
- *mshrt* - The record corresponds to user collected data of the *short* type for C.
- *open* - The record corresponds to a *copen* event.
- *probe* - The record corresponds to a *probe* event.
- *recv* - The record corresponds to a *recv* event.
- *recvw* - The record corresponds to a *recvw* event.
- *rdled* - The record corresponds to a *redled* event.
- *send* - The record corresponds to a *send* event.
- *sendw* - The record corresponds to a *sendw* event.
- *stat* - The record corresponds to a *status* event.
- *slog* - The record corresponds to a *syslog* event.
- *wrapr* - This record says that the count for the message type received, node received from, and pid received from has wrapped around to 1. This is mainly used for correlation of send and receive records.

- *wraps* - This record says that the count for the message type sent, node sent to, and pid sent to has wrapped around to 1. This is mainly used for correlation of send and receive records.

- Field 2 (2 byte integer)

  - All Record Types - This field contains the node number of the recording process.

- Field 3 (2 byte integer)

  - All Record Types - This field contains the pid number of the recording process.

- Field 4 (4 byte integer)

  - *end* - For this type, the field will contain a pseudo global time that corresponds to an approximate end time as collected by the end of the PRASEEND routine (this time is recorded when the ending routine is entered). For this record type, the time listed here will be the same as the time listed in Field 5.

  - *init* - For this type, the field will contain a pseudo global time that corresponds to an approximate end time of the initiation software contained in PRASE.

  - *wrapr* - For this type, the field will contain a pseudo global time that corresponds to a time just prior to the associated receive event that caused the count to wrap.

  - *wraps* - For this type, the field will contain a pseudo global time that corresponds to a time just prior to the associated send event that caused the count to wrap.

- All Other Record Types - This field contains a pseudo global time refering to just prior to the initiation of the event.

- Field 5 (4 byte integer)

  - *dump* - For this type, this field contains a pseudo global time that corresponds to a time near to the end of the dump routine. The actual dump of data will already have a occurred when the time was recorded. However, if you are only allowing one record to be in the trace at a time, then a message containing the dump record will be sent after this time (you have to get the time for the record prior to sending the record).

  - *end* - For this type, this field contains a pseudo global time that corresponds to the time that the end routine was entered. For this record type, this time will be the same as the time given in Field 4.

  - *init* - For this type, this field contains a pseudo global time that corresponds to a time near to the end of the initialization routine. Synchronization has already occurred at this point. However, if you are only allowing one record to be in the trace at a time, then a message containing the init record and one containing a dump record will be sent after this time (you have to get the time for the record prior to sending the record).

  - *mchar, mdble, mflot, mint, mlong, mshrt* - For these types of records, this field contains a pseudo global time that corresponds to a time near to the end of the routine that records the user information. However, if you are only allowing one record to be in the trace at a time, then a message containing the user record and one containing a dump record will be sent after this time (you have to get the time for the record prior to sending the record).

  - *recv* - For this type, this field can have a couple of different meanings. If the *recv* completes by the time a status is performed by PRASE on the

call to *recv*, then this is the pseudo global time right after we exit the call to the Hypercube *recv* call. If, however, the initial status shows that the buffer is not yet clear, this time will reflect the pseudo global time right after the call to the Hypercube *status* call that shows that this is clear.

- *send* - For this type, this field contains a pseudo global time that corresponds to a time just after the completion of the *send*. This does not mean that the buffer has been cleared. It only means that the *send* call was completed.
- *wrapr* - For this type, a pseudo global time is recorded. The time carries the same meaning as the associated receive event. If a *recv* follows this record, then look at the explanation for the *recv* service for the meaning of this field. If a *recvw* follows this record, then look at the explanation for the *recvw* service for the meaning of this field.
- *wraps* - For this type, a pseudo global time is recorded. The time carries the same meaning as the associated send event. If a *send* follows this record, then look at the explanation for the *send* service for the meaning of this field. If a *sendw* follows this record, then look at the explanation for the *sendw* service for the meaning of this field.
- All Other Record Types - This field contains a pseudo global time refering to just after the completion of the event.

- Field 6 (Variable length)
  - *close, open, probe, recv, recvw, send, sendw, stat* - For these types, this field holds the channel number that the service dealt with (2 Byte Integer).
  - *dump* - For this type, this field holds the channel number on which the dump was performed. This particular channel is opened by PRASE to use for its own communication separate from the user's program(s) (2 Byte Integer).

- *mchar* - For this type, this field holds character information as recorded by the user. There are 13 characters of information and one character for a null at the end. (14 Characters total)
- *mdble* - For this type, this field holds the user supplied information as recorded by the user. (This is the size of a double in C on the Intel Hypercube)
- *mflot* - For this type, this field holds the user supplied information as recorded by the user. (This is the size of a float in C on the Intel Hypercube)
- *mint* - For this type, this field holds the user supplied information as recorded by the user. (This is the size of an integer in C on the Intel Hypercube)
- *mlong* - For this type, this field holds the user supplied information as recorded by the user. (This is the size of a long in C on the Intel Hypercube)
- *mshrt* - For this type, this field holds the user supplied information as recorded by the user. (This is the size of a short in C on the Intel Hypercube)
- *wrapr, wraps* - For these types, this field holds the channel number for the service that corresponds to this wrap (2 Byte Integer).
- All Other Record Types - For these types, this field is not pertinent (2 Byte Integer).

- Field 7 (2 byte integer)
  - *dump* - For this type, this field holds the PRASE message type used to dump information.

- *grled* - For this type, this field holds the state passed to the *greenled* service.
- *mchar, mdble, mflot, mint, mlong, mshrt* - For these types, this field does not exist.
- *probe, recv, recvw* - For these types, this field holds the message type that the service dealt with.
- *rdled* - For this type, this field holds the state passed to the *redled* service.
- *send, sendw* - For these types, this field holds the message type that the service dealt with.
- *stat* - For this type, this field holds the result from the call to *status*.
- *wrapr, wraps* - For these types, this field holds the message type for the service that corresponds to this wrap (2 Byte Integer).
- All Other Record Types - For these types, this field is not pertinent.

• Field 8 (2 byte integer)

- *dump* - For this type, this field holds the amount of bytes that were dumped.
- *mchar, mdble, mflot, mint, mlong, mshrt* - For these types, this field does not exist.
- *probe* - For this type, this field holds the result of the call to the *probe* service.
- *recv, recvw* - For these types, this field holds the number of bytes returned from the service.
- *send, sendw* - For these types, this field holds the number of bytes passed into the service in the parameter telling the service how many bytes to send.

- *wrapr* - For this type, this field holds the same value that the corresponding receive event holds.
- *wraps* - For this type, this field holds the same value that the corresponding send event holds.
- All Other Record Types - For these types, this field is not pertinent.

• Field 9 (2 byte integer)

- *dump* - For this type, this field reflects the number of times a dump has occurred.
- *mchar, mdble, mflot, mint, mlong, mshrt* - For these types, this field does not exist.
- recv, recvw - For these types, this field holds a count from the beginning of the run that is sequential pertaining to five specific attributes of a message. The five attributes are, the recording node, recording pid, associated node, associated pid, and message type. If the these all match (the first two will since this is the local node and pid), then the count is incremented and included in this field. A separate set of these counts are kept for messages being sent from this node. The counts discussed here pertain only to messages received.
- send, sendw - For these types, this field holds a count from the beginning of the run that is sequential pertaining to five specific attributes of a message. The five attributes are, the recording node, recording pid, associated node, associated pid, and message type. If the these all match (the first two will since this is the local node and pid), then the count is incremented and included in this field. A separate set of these counts are kept for messages being received from this node. The counts discussed here pertain only to messages sent.

- *wrapr* - For this type, this field holds the same value that the corresponding receive event holds.
- *wraps* - For this type, this field holds the same value that the corresponding send event holds.
- All Other Record Types - For these types, this field is not pertinent.

• Field 10 (2 byte integer)

- *dump* - For this type, this field holds the address of the host.
- *mchar, mdble, mflot, mint, mlong, mshrt* - For these types, this field does not exist.
- *recv, recvw* - For these types, this field holds the node that the message was received from.
- *send, sendw* - For these types, this field holds the node that the message was sent to.
- *wrapr* - For this type, this field holds the same value that the corresponding receive event holds.
- *wraps* - For this type, this field holds the same value that the corresponding send event holds.
- All Other Record Types - For these types, this field is not pertinent.

• Field 11 (2 byte integer)

- *dump* - For this type, this field holds the pid of the host process that the data is being sent to.
- *mchar, mdble, mflot, mint, mlong, mshrt* - For these types, this field does not exist.
- *recv, recvw* - For these types, this field holds the pid of the process that the message was received from.

- *send, sendw* - For these types, this field holds the pid of the process that the message is being sent to.
- *wrapr* - For this type, this field holds the same value that the corresponding receive event holds.
- *wraps* - For this type, this field holds the same value that the corresponding send event holds.
- *slog* - For this type, this field holds the pid passed into the *syslog* service.
- All Other Record Types - For these types, this field is not pertinent.

• Field 12 (2 byte integer)

- *mchar, mdble, mflot, mint, mlong, mshrt* - For these types, this field does not exist.
- All Other Types - This field holds a -1 at this point in time and has been included mainly for compatibility with Seecube.

An example of this record format would be as follows:

```
sendw   0   4   3480   3492   139   10   256   1   18   7   -1
```

Appendix D. *Obtaining Information about Seecube*

If you are interested in getting more information about Seecube; or you want to obtain a copy for yourself, the following information may be helpful.

Mr. Alva Couch (the author of Seecube - who by the way has been very helpful throughout the effort) should be contacted at Tufts University. His address is:

Alva Couch
Department of Computer Science
Tufts University
Medford Mass. 02155

Phone: (617) 381-3674

CSNET ADDRESS: couch@cs.tufts.edu

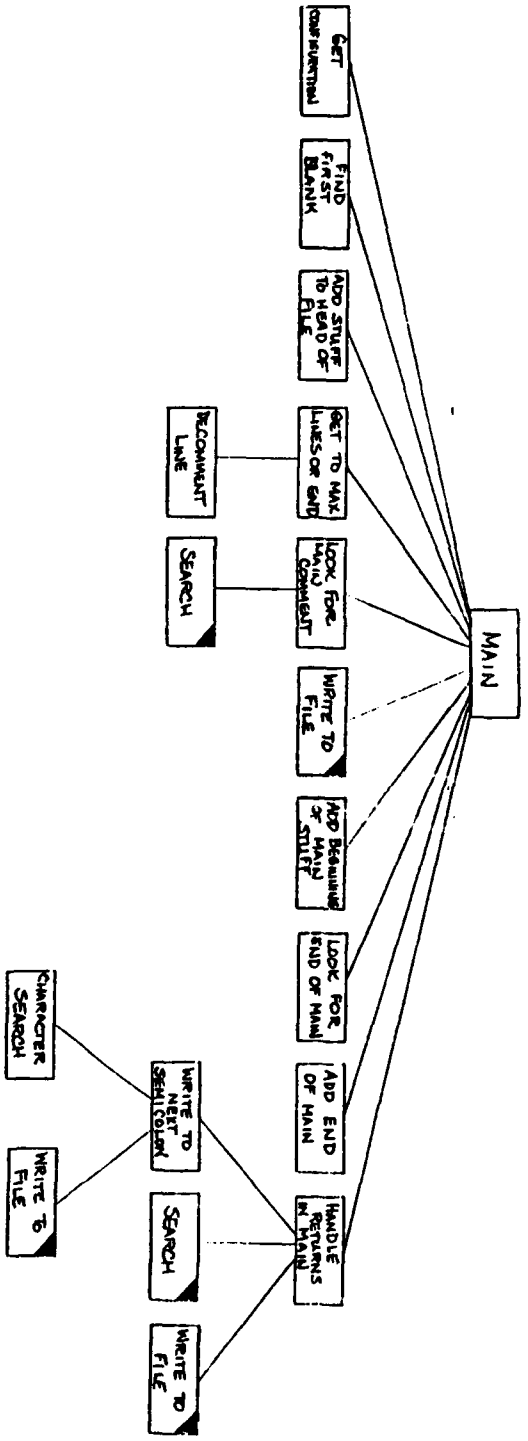
Seecube will currently work with an NCUBE Hypercube as well if you have some interest in that particular machine.

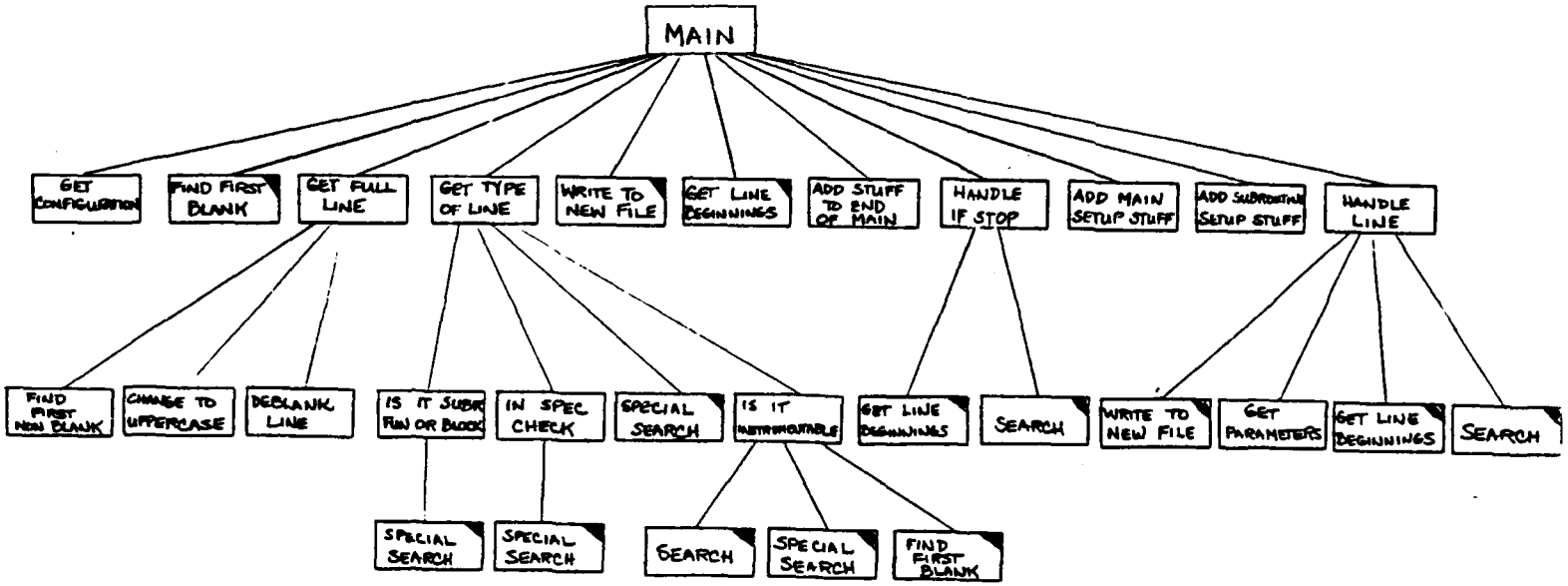
Appendix B. *Structure Charts and Pseudocode*

The following pages include basic structure charts for the two preprocessors, and the translator. Pseudocode is included for the main modules only of the preprocessors and the translator. Pseudocode for the prase_clct routine and for the data collection synchronization subroutine are also provided. No other pseudocode is provided.

Throughout the structure charts, certain modules are called multiple times from a parent routine. Only one instance of the subroutine is shown.

prase_c_prc.a STRUCTURE CHART



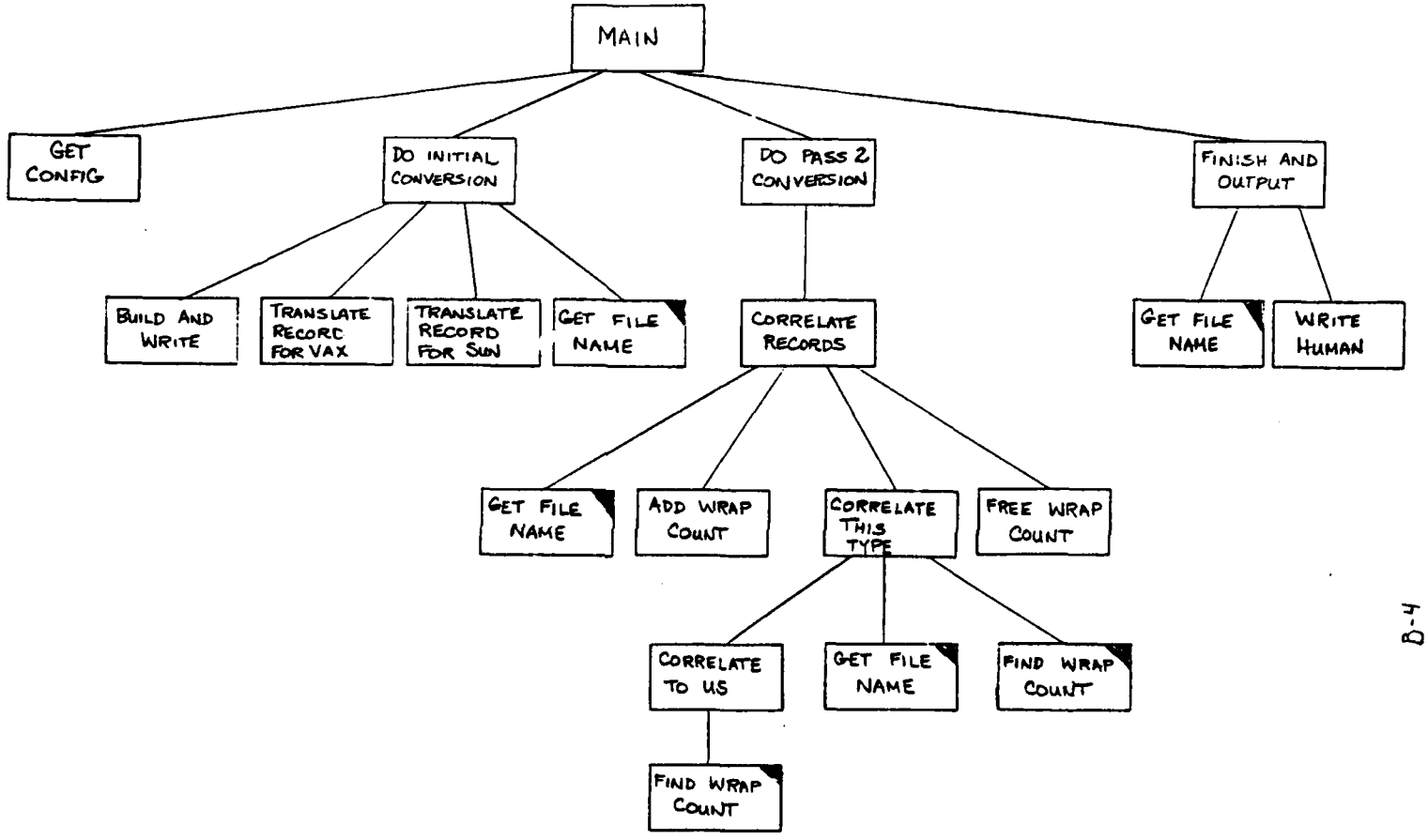


B-3

prase_tor_pre.a
STRUCTURE CHART

NOTE: SPECIAL SEARCH CALLS SEARCH EVEN THOUGH IT IS NOT SHOWN IN THE DIAGRAM ABOVE

prase_trans.a STRUCTURE CHART



Pseudocode for prase_C_pre.a
(Main Routine only)

```
Initialize variables
Get configuration information
For each file loop
    Reset variables
    Setup file names and open files
    Add code to head of output file
    Reset more variables
    Get to max lines or end (Gets as many lines as
                             it can up to max. Returns the
                             original lines as well as a corresponding
                             version of the lines with comments
                             removed)
    While not done with the file, loop
        If not done with main and not in main
            Look for main "comment"
            If found main statement
                In_main = true
                Write to file up to the end of main
                    "comment" statement
                Add beginning of main code
                Brace_depth = 1
                Get to max lines or end
            End if
        End if
```

```
If in main
    Look for end of main
    If found end of main
        Done with main = true
        In_main = false
        If we are at the end of main already
            Add end of main code
            Write to end of main position
        Else
            Handle returns in main (prior to returns in the main routine
            insert calls to the ending code)
            (returns in main are considered
             the same as exits)

            Prior to the end of main add
            a call to the ending code

        End if
    Else
        Set Search_to_position = end of last
            line read in
    End if
    Handle returns in main
Else
    Write all lines to file
End if
```

```
            Get to mark lines or end
            If end of file found
                  If line list is empty set done with file = true
                  End if
            End if

      End loop
      Close files
End loop
```

Pseudocode for prase_clct.c
(Main Routine Only)

```
Open a channel for receiving messages
Do initialization
Loop forever
    Issue a receive and wait
    Calculate how many records received
    Build file name
    Open file
    Write records
    Close file
End loop
```

Pseudocode for the data collection synchronization routine — no other data collection pseudo code has been provided

Get start_time

Do some initialization

Determine if our pid is the first pid in the pid's array - if so, first_pid should = true

Determine if we are on the lowest node number running code - if so, lowest_node should = true

If we are the lowest_node and first_pid (this means we are the controlling process (CP))

For each possible node loop

If the node we are investigating is not my node and it is running code then

Tell the first_pid on the node that it is time to synchronize

Issue a receive and wait

Get the current time

Send the time back to the node

End if

End loop

PRASE_subtract_time = start time (PRASE_subtract_time can be subtracted from the clock time (after synchronization) to get a global time stamp)

Tell other pids on our node our start time

Else if we are not the lowest_node but we are the first_pid

Issue a receive and wait (we are waiting for CP to tell us to synchronize)

Get a send_time

Issue a send to CP

Issue a receive and wait

Get a receive_time

Trans_time = (receive_time − send_time) / 2

Correlate_time = send_time + trans_time

Figure a difference between our clock and the CP's node clock using correlate_time minus the time CP sent to us—these 2 times are assumed to represent approximately the same absolute time (diff_time)

PRASE_subtract_time = diff_time − CP's start_time
(CP's start time was passed in the message)

Tell other pids on our node CP's start_time and our PRASE_subtract_time

Else we are not the first pid

Issue a receive and wait

PRASE_subtract_time = our node's passed PRASE_subtract_time or if we are the lowest node, the start time that was passed

End if

Pseudocode for prase_for_pre.a

```
Initialize variables
Get configuration information
For each file loop
    Setup file names and open files
    Reset variables
    While there are more lines in this file, loop
        Get a full logical statement (which may
          encompass several physical lines to
          get continuation lines) - return
          statement type if Known
        If returned type is Known, assign it
          to this line's type
        Else get type of line
        End if
        If this line's type = comment
            Write lines to new file
        Else if this line's type = specification
            Write lines to new file
        Else if this line's type = subroutine function
                or block data statement
            In specification statements = true
            In main routine = false
```

Else if this line's type = end statement

If in the main fortran routine

Add ending code

Else write to new file

End if

If we haven't processed the main routine yet

Assume to be in main until proven wrong

Set in specification = true

End if

Else if this line's type = stop

Add ending code

Write out stop

Else if this line's type = if_stop (this corresponds to a logical if structure with a stop being executed if true)

Change if to an if-then structure

Add ending code

Write stop statement

Else if this line's type = empty line

Write lines to new file

```
Else
    If in specification = true
        Set in specification = false
        If in main then add main setup
            code
        Else
            Add subroutine setup code
        End if
    End if
    Handle this line
End if

End loop
Close files

End loop
```

Pseudocode for prase_trans.a
(main routine only)

Initialize global variables

Get configuration file information

Do initial conversion (convert all fields
except human file fields 13, 14, and
4 for copens)

Do pass 2 conversion (correlate fields 12+13
in the newly built seecube records)

Finish and output

Report any uncorrelated sends and receives

Appendix C. *System Configuration Guide*

The purpose of this appendix is to convey to the reader how the system is configured. Specifically, where do files reside, what are they for, and what do they look like.

First, the overall structure of the system. Certain Microvax's and Sun workstations are all able to directly access the same disk. For these two types of machines, a single disk structure has been constructed for PRASE. A separate structure exists for the related Seecube files.

PRASE has a system level directory called /usr/PRASE. Under this directory there are three subdirectories and a file. The file is a default copy of the configuration file. This file will get copied into a user's directory when the **prase_new** command is executed. There is a *bin* subdirectory which holds files used for initial setup. There is a second subdirectory called *sunbin*. This holds any PRASE executables that run on a Sun workstation. A third subdirectory holds executables that run on a Microvax using the Ultrix Operating System. This subdirectory is called *vaxbin*. The *setup* procedures, when executed, set up the appropriate commands to allow a user to run the correct code for the machine that he is using.

Still on the Sun/Microvax disks, there is a /usr/Seecube system level directory. There are also three subdirectories and a file located in this directory. The file *context*, and the subdirectories *clumps* and *colors* are copied into a user's directory when the **prase_new** (see the User's Manual for details) command is executed. These provide defaults used by Seecube. The *bin* subdirectory holds the *resolve* and *seecube* executables.

On the Hypercubes, there is one system level directory called /usr/PRASE . Under this directory, there are two subdirectories. The *bin* directory holds two setup type files, and two executables. The two executables are the prase_clct and prase_view programs. There is another subdirectory called *source*. This holds source files that will need to be copied into a user's directory.

The following is a type of system configuration guide. The items to the far left are the system level directories. As you move to the right on the page, you come to subdirectories or files until there are no more levels. Source code, which is not needed by the user, will be kept elsewhere in a repository. The following listing is mainly for items a user will need to access.

Sun/Microvax Environment

```
/usr/PRASE
        bin/
            prase_new*        (Used to set up a new area
                               for using the PRASE and
                               Seecube software.)
            prase_sunsetup*   (Sets up aliases for the
                               user in a Sun environment.
                               It automatically
                               points the user to the
                               correct files.)
            prase_vaxsetup*   (Sets up aliases for the
                               user in an Ultrix Microvax
                               environment.  It automatically
                               points the user to the
                               correct files.)

        prase.cfg  (This is the default configuration
                    file that gets copied into a user's
                    directory when the prase_new command
                    is executed.)
```

```
            sunbin/
                (currently empty)

            vaxbin/
                vax_prase_c_pre*
                vax_prase_for_pre*
                vax_prase_trans*

/usr/Seecube
            bin/
                resolve* (executable)
                seecube* (executable)

            clumps/  (This directory holds files that are
                      copied for the user's use of seecube.)
                2-per-clump
                4-per-clump
                8-per-clump

            colors/  (This directory holds subdirectories that
                      hold files for the user's use of seecube.)
                4_only/
                     cmap_cload
                     cmap_csize
                     cmap_eload
                     cmap_esize
                     cmap_nload
                     cmap_nsize
```

```
            default/  (The rest of these subdirectories
                       contain files with the same names
                       as in the 4_only subdirectory.)
            graduated/
            light_color/
            logarithmic/
            test/
            weird/

          context  (This is a file that holds defaults
                    that seecube uses for initial
                    display information - copied for
                    user.)
```

iPSC/1 Hypercube Environment

```
  /usr/PRASE
            bin/
                prase_clct* (This is the background collection
                             routine that runs on the host.)
                prase_newcube*  (Used to set up a new area for
                                 using the PRASE software.)
                prase_setup*  (Sets up aliases for the
                               user in an Hypercube environment.
                               It automatically points the
                               user to the correct files.)
```

```
        prase_view*  (This allows the user to view
                      the data in the binary files -
                      this is the program run by the
                      pv command.)

    source/
        PRASE_FOR.H     (Fortran header file - copied
                         for user)
        prase_code.c    (PRASE Subroutines - copied for
                         user)
        prase_extern.h (C header file - copied for
                         user)
        prase_glob.h    (C header file - copied for
                         user)
        prase_incl.h    (C header file - copied for
                         user)
        prase_user.h    (C header file - default copy
                         copied for user)
```

The other source code files that are pertinent to the system but that have not been listed are:

```
prase_for_pre.a     (Fortran Preprocessor source)
prase_c_pre.a       (C Preprocessor source)
prase_trans.a       (Translator source)

prase_clct.c        (Source code for the program
                     that runs in the background on
                     the Hypercube that accepts data
                     from the nodes.)
```

`prase_view.c`	(Source code for the program that takes a binary PRASE data file on the Hypercube and displays the contents in ASCII. This program only runs on the Hypercube. It will not run properly on a Sun if compiled there.)
`Seecube source`	(Several Seecube source files and other files are also kept either in the repository or on tape.)

The rest of this appendix holds certain copies of files pertinent to the configuration and rebuilding of the PRASE system. Note that some of the comments in the files have been rearranged for printing. The words, however, have not changed.

There is a Sun setup procedure that should be run if a user is on a Sun workstation. The procedure follows:

File Name and Path: /usr/PRASE/bin/prase_sunsetup

```
#     PRASE Setup for the Sun.
#
# Author: Capt Mark Kahl
#
#   Date: 4 November 1988
#
# This file will set up the stuff a user needs to use the PRASE
# system on a Sun Workstation.  Things are different between the
# Vax and the Suns, so someone running on the Sun should use this
# setup file.  To move or reinstall the PRASE or Seecube software,
# you should change the path names in this file to reflect the
# appropriate information.
#
#
#
# --First we set up some aliases so that the user can get to
#   the files he needs.
#
alias seecube '/usr/Seecube/bin/seecube'
alias resolve '/usr/Seecube/bin/resolve'
alias prase_trans '/usr/PRASE/sunbin/prase_trans'
alias prase_c_pre '/usr/PRASE/sunbin/prase_c_pre'
alias prase_for_pre '/usr/PRASE/sunbin/prase_for_pre'
#
#
# --Now we do one last alias that allows the user to copy
#   in all the stuff that he will need to run with.  This
#   includes a default configuration file as well as all
#   the Seecube stuff needed.  It also makes a data area
#   to place the data in.
#
alias prase_new 'source /usr/PRASE/bin/prase_new'
#
```

There is also a Vax setup procedure that should be run if a user is on a Microvax. This file follows:

File Name and Path: /usr/PRASE/bin/prase_vaxsetup

```
#       PRASE Setup for the vax (olympus).
#
# Author: Capt Mark Kahl
#
#   Date: 2 November 1988
#
# This file will set up the stuff a user needs to use
# the PRASE system on the Vax (olympus).  Things are
# different between the Vax and the Suns, so someone
# running on the Vax should use this setup file.  To
# move or reinstall this software, you should change
# the path names in this file to reflect the current
# information.
#
#
#
# --First we set up some aliases so that the user can get to
#   the files he needs.
#
alias prase_trans '/usr/PRASE/vaxbin/vax_prase_trans'
alias prase_c_pre '/usr/PRASE/vaxbin/vax_prase_c_pre'
alias prase_for_pre '/usr/PRASE/vaxbin/vax_prase_for_pre'
#
#
# --Now we do one last alias that allows the user to copy
#   in all the stuff that he will need to run with.  This
#   includes a default configuration file as well as all
#   the Seecube stuff needed.  It also makes a data area
#   to place the data in.
#
alias prase_new 'source /usr/PRASE/bin/prase_new'
#
```

The next file sets up a new directory by copying things for the user. This file can be run from either a Sun or Microvax.

File Name and Path: /usr/PRASE/bin/prase_new

```
#       PRASE new to setup for a new PRASE area.
#
# Author: Capt Mark Kahl
#
#   Date: 2 November 1988
#
# This file copies in default items such as a default
# configuration file, and all the seecube support stuff
# needed.  To move or reinstall this software, you
# should update this file to show current information.
#
#
#
# --Ok, now we copy in the stuff.
#
echo " "
echo " "
echo "Copying files . . ."
echo " "
#
cp /usr/PRASE/prase.cfg .
cp /usr/Seecube/context .
cp -r /usr/Seecube/colors .
cp -r /usr/Seecube/clumps .
#
#
echo "Complete . . ."
echo " "
#
#
```

There are two files that are used on the Hypercubes for setup. First the setup file.

File Name and Path: /usr/PRASE/bin/prase_setup

```
# PRASE Setup for the Hypercubes.
#
# Author:  Capt Mark Kahl
#
#   Date:  4 November 1988
#
# This file will set up the stuff a user needs to use the
# PRASE software.  This will set up aliases so that a user
# can get access to any of the PRASE code the he wants.  To
# move the code or to reinstall the software, just update
# the pathnames below.
#
#
#
# --We will set up the aliases now for running the programs
#   the user will need to be able to run.
#
alias prase_clct '/usr/PRASE/bin/prase_clct&'
alias pv '/usr/PRASE/bin/prase_view'
#
#
#
# --Now we give the user the capability to make a new
#   prase_code.o file without having to know the command.
#
alias pcode_compile 'cc -c -Alhu -K prase_code.c'
#
#
#
# --Now we provide the user with one last alias.  This will
#   run a script that copies all of the PRASE files into a
#   users directory.  It will also create a data directory
#   beneath the user's current directory.
#
alias prase_newcube 'source /usr/PRASE/bin/prase_newcube'
```

The second file on the Hypercubes sets up a directory for a new session of using PRASE as well as copying in needed files.

File Name and Path: /usr/PRASE/bin/prase_newcube

```
# PRASE newcube to setup for a new PRASE are.
#
# Author:  Capt Mark Kahl
#
#   Date:  4 November 1988
#
# This file copies in default items needed for a PRASE session.
# It also creates a data directory beneath the current default
# directory.  To relocate the files or reinstall the software,
# just change the pathnames below to reflect the correct
# configuration.
#
#
#
# --Ok, now we copy in the stuff.
#
echo " "
echo "Copying files into your default directory."
echo " "
#
cp /usr/PRASE/source/PRASE_FOR.H .
cp /usr/PRASE/source/prase_code.c .
cp /usr/PRASE/source/prase_glob.h .
cp /usr/PRASE/source/prase_user.h .
cp /usr/PRASE/source/prase_extern.h .
cp /usr/PRASE/source/prase_incl.h .
#
#
#
# --Now we make the directory for data.
#
echo " "
echo "Creating a data directory called prase_data."
echo " "
#
mkdir prase_data
```

Now the makefile for the two preprocessors is included. It really is just a script. No dependencies are used.

```
prase_for_pre:
ada -M prase_for_pre.a -o prase_for_pre

prase_c_pre:
ada -M prase_c_pre.a -o prase_c_pre
```

Now the makefile is included for the translation program. This program should be able to be compiled on either the Sun or Microvax. However, you will have to do some commenting of a particular *if* statement. If you get an error on compilation (not just a warning), go to that line in the code and read the instructions around the errored line. If the error stems from being on a certain machine that doesn't match with the correct if structure, just comment and uncomment as necessary.

```
prase_trans:
ada -M prase_trans.a -o prase_trans
```

A makefile for the prase_clct.c and prase_view.c programs on the Hypercubes is also provided.

```
#  This make file should be used to make the programs prase_clct and
#  prase_view.
#
#

CFLAGS = -Alhu -K

all: prase_clct prase_view

help:
@echo " "
@echo " "
@echo "make all or make    - makes prase_clct and prase_view"
@echo "make prase_clct     - makes the collection program that"
@echo "                      runs in the background on the host."
@echo "make prase_view     - makes the viewing program that allows"
@echo "                      a user to view the data in the files"
@echo "                      holding the collection results."
@echo " "
@echo " "


prase_clct: prase_clct.c
cc -Alhu -o prase_clct prase_clct.c /usr/ipsc/lib/chost.a

prase_view: prase_view.c
cc -Alhu -o prase_view prase_view.c
```

Appendix D. *Volume II Pointer for Test Cases and Results*

The test cases and results have not been incorporated into this document. It has been placed in Volume II. This appendix is being used as a pointer to where the results may be found. For information pertaining to the test cases and results associated with this thesis, contact:

Dr. Thomas C. Hartrum
Dept. of Electrical & Computer Engineering
Air Force Institute of Technology
AFIT/ENG
Wright-Patterson AFB, OH 45433-6583

Appendix E. *Seecube File Formats*

One goal of this thesis was to provide compatibility with the local version of the Seecube software. A translation program has been provided that will take the new data format and translate a data file into the Seecube data format. The main thrust of this appendix, is to briefly discuss the current Seecube data format. For more information, refer to the Seecube User's Manual [3:4-11]. All that is really being discussed here is the author's understanding of the fields and the fields of interest to the author. No attempt is made to cover every possible field entry allowed by Seecube.

The Seecube version that AFIT has obtained, requires data formatted in a specific way to be useable by the Seecube Resolver. The data file name must be named *human*, and the file must be in ascii. Other data files are produced when the Resolver is run, but these will not be discussed here since they now become specific to the Seecube system (nothing that must be interfaced with by PRASE directly).

The Seecube data format contains several fields, and in a general sense, provides a format that is usable in many different situations. It appears that Seecube has two records that relate to the *sendw* and *recvw* utilities. One record lists the beginning of the call and the other record pertains to the end of the call. It seems that the major differences between the two records are the time and cross-referencing information.

Seecube's Resolver accepts a file containing records with 14 fields. To aid in the discussion, two example records and explanations pertaining to these records are provided.

Record Pair Example

```
0  4  5  6  3480  wb  139  10  256  18  7  18  17  x
0  4  6  5  3492  we  139  10  256  18  7  18  17  x
```

- Field 1 - This field contains the node number that the event was collected on. (0 in the example)

- Field 2 - This field contains the *pid* (process id) of the process that collected the data. (4 in the example)
- Field 3 - This field contains a number corresponding to the event. This apparently is a sequential number. Each consecutive event gets the next sequential number. (5 in the first record of the example - this means that this record was the sixth event recorded – events start numbering at 0)
- Field 4 - This field contains a reference number to the corresponding event on the same node. In the example, this field points to the next record which is the corresponding write end. Notice that the second record points back to the write begin.
- Field 5 - This field contains a pseudo global time. (3480 in record 1)
- Field 6 - This field holds what Couch calls the "opcode" field [3:8-9]. It identifies the type of record. Field values mean different things based on what this field contains.
- Field 7 - This field holds the number of the communications channel that the event occurred on. (139 in both records)
- Field 8 - This field contains the message type. (10 for both records)
- Field 9 - This field holds the message size. (256 in both records)
- Field 10 - This field contains the node that the message is being sent to. (18 in the example)
- Field 11 - This field contains the *pid* that the message is being sent to. (7 in the example)
- Field 12 - This field contains the sequential event number logged on the associated node that corresponds to the event logged on the node recording this record.

- Field 13 - This field contains the companion event (for our field 12) on the associated node.
- Field 14 - This field contains what Couch calls a 'text marker' [3:8]. It is an integer and can be used to uniquely identify each calling location within a piece of code. For example, each *sendw* can be coded with a different marker integer so that the trace records can be correlated with a specific message call. In our example, this field contains an x because gathering of this field is evidently not implemented in the version of Seecube that we possess.

To restate an important point, this seems to be a general format that can take several things into account. Also, the cross-referencing information is sent along with the message making the whole data collection format and collection method somewhat resistant to errors (if an error occurs, you can throw away the bad record but still have the other good records around).

Another point that should be made is that the main version of the Seecube Data Collector runs on the NCUBE Hypercube. That may be another reason for the file format being used. The example has been given in light of how I understand the records to fit with the message events caused by calls to the Hypercube routines.

Appendix F. *PRASE File Formats*

There are four file formats that PRASE deals with. One is the format of the configuration file. An example of this file is provided in its own appendix in the User's Manual. There are also discussions of the file and its use in other parts of the User's Manual and the main body of the Thesis. Therefore, no further discussion will be included here.

A second is the file that contains the data collected on the Hypercube for a run. This file has been discussed in Appendix C of the User's Manual. It is discussed there in reference to the **pv** command but the fields are explained in sufficient detail. It should be noted that each record in that file requires 32 bytes of storage and is stored on disk in binary. It appears that depending on which machine the file is being read on, it must be read differently (or least using a different format). The data is stored on the Hypercube with the low data byte being stored first.

The third is a file that the translator uses as an intermediate file for processing. This file will be discussed here. Finally, the translator must put out the file format that is used by the Seecube Resolver. This file format has been discussed in the appendix specific to the Seecube file format.

F.1 Intermediate Translation File

The translator does not change the original data files containing the data collected on the Hypercubes. Instead, it keeps intermediate results in temporary files. The files are named the same as the data files except that a 't' is added to the beginning of the name. An intermediate file name for node 5 *pid* 7 would be tn5_p7

.

The files are written out using a construct in Ada and then read back in. They are, however, not deleted from the disk. Actual physical formats are unknown especially between machines (Sun and Microvax). The formats are very possibly different between the machines. I will provide only the format as defined in the Ada source code. It should be noted that unless the translator runs to completion, the

files may be in some state other than a final state that reflects the following data. The first 12 fields have the same meanings as the data collection files as outlined in Appendix C of the User's Manual. The only difference is that there are a limited number of record types allowed in this file. The types are provided in the first field description. Other fields (that do not have a corresponding field in the data collection fields) are explained. The entire record is 53 bytes long.

It follows:

- Field 1 (char 6) - This field can contain one of the following record types: *close, open, recv, recvw, send, sendw, wrapr,* and *wraps.*
- Field 2 (2 byte integer) - recording node.
- Field 3 (2 byte integer) - recording pid.
- Field 4 (4 byte integer) - begin time.
- Field 5 (4 byte integer) - end time.
- Field 6 (2 byte integer) - channel.
- Field 7 (2 byte integer) - message type (if applicable).
- Field 8 (2 byte integer) - message size (if applicable).
- Field 9 (2 byte integer) - message count (if applicable).
- Field 10 (2 byte integer) - corresponding node (if applicable).
- Field 11 (2 byte integer) - corresponding pid (if applicable).
- Field 12 (2 byte integer) - Seecube compatible marker.
- Field 13 (char 2) - Seecube begin record designator. (one PRASE record may often translate into two Seecube records - this is the "opcode" [3:8-9] for the beginning record)
  - *close* - For this type, the field will contain a 'cc'.

- *open* - For this type, the field will contain a 'co'.
- *recv, recvw* - For this type, the field will contain a 'rb'.
- *send, sendw* - For this type, the field will contain a 'wb'.
- *wrapr* - For this type, the field will contain a 'wr'. However, this field does not correspond to a Seecube "opcode" and is not translated into the *human* file.
- *wraps* - For this type, the field will contain a 'ws'. However, this field does not correspond to a Seecube "opcode" and is not translated into the *human* file.

- Field 14 (char 2) - Seecube end record designator. (one PRASE record may often translate into two Seecube records - this is the "opcode" [3:8-9] for the ending record)
  - *close* - For this type, the field will contain blanks.
  - *open* - For this type, the field will contain blanks.
  - *recv, recvw* - For this type, the field will contain a 're'.
  - *send, sendw* - For this type, the field will contain a 'we'.
  - *wrapr* - For this type, the field will contain blanks. However, this field does not correspond to a Seecube "opcode" and is not translated into the *human* file.
  - *wraps* - For this type, the field will contain blanks. However, this field does not correspond to a Seecube "opcode" and is not translated into the *human* file.

- Field 15 (4 byte integer) - This field holds the value that will ultimately be placed in the *human* file field number 3.

  - *close, open, recv, recvw, send, sendw* - For these types, this field will contain a value that corresponds to the *human* file field number 3.

  - *wrapr, wraps* - For these types, this field will contain a count pertaining to a previous record that was not a *wrapr* or *wraps*. In the case of previous records of the types *recv, recvw, send* and *sendw*, this will point to the begin count (or Field 15).

- Field 16 (4 byte integer) - This field holds the value that will ultimately be placed in the *human* file field number 4.

  - *close, open, recv, recvw, send, sendw* - For these types, this field will contain a value that corresponds to the *human* file field number 4.

  - *wrapr, wraps* - For these types, this field will contain a count pertaining to a previous record that was not a *wrapr* or *wraps*. In the case of previous records of the types *recv, recvw, send* and *sendw*, this will point to the begin count (or Field 15).

- Field 17 (4 byte integer) - This field holds the value that will ultimately be placed in the *human* file field number 12.

  - *close, open* - For these types, this field is undefined.

  - *recv, recvw, send, sendw* - For these types, this field will contain a value that corresponds to the *human* file field number 12. If a record is not correlated, the value will be 0.

  - *wrapr, wraps* - For these types, this field is undefined.

- Field 18 (4 byte integer) - This field holds the value that will ultimately be placed in the *human* file field number 13.

- *close, open* - For these types, this field is undefined.
- *recv, recvw, send, sendw* - For these types, this field will contain a value that corresponds to the *human* file field number 13. If a record is not correlated, the value will be 0.
- *wrapr, wraps* - For these types, this field is undefined.

- Field 19 (1 byte boolean) - This field will be true if a *recv, recvw, send,* or *sendw* record has been correlated with the appropriate corresponding record.
  - *close, open* - For these types, this field is undefined.
  - *recv, recvw, send, sendw* - This field will be true if the record has been correlated with its counterpart record.
  - *wrapr, wraps* - For these types, this field is undefined.

Appendix G. *File Translation Discussion*

This thesis effort has resulted in providing compatibility with the version of the Seecube software available here at AFIT. A translation program has been provided that will take the new data format and translate the data files into the Seecube data format. The main thrust of this appendix is to show that the existing Seecube data format can indeed be derived from the new data format.

Two reasons for a new format include 1) less user interaction was desired (no need to change messages etc.) and 2) space savings seemed possible. This is not meant to imply that Seecube was done incorrectly or even inefficiently; on the contrary, Seecube appears to have been implemented with a very general format which seems applicable in several situations. However, it seems that the work can be done differently while still maintaining compatibility with Seecube.

The Seecube data format contains several fields, and in a general sense, provides a format that is usable in many different situations. AFIT's local copy of Seecube requires messages to contain empty bytes at the beginning. This is used to send cross-referencing information to the receiving node. This cross-referencing information can then be used after the run to correlate the sending records from one node with the receiving records on another node. This is important because a user may then get some idea as to when a message's transmission began and ended. With the new data format, these cross referencing fields can be derived from the ordering on each individual node and from the ordering imposed when sending messages of the same type within the Intel iPSC environment. Thus, the new format does not require a user to keep the first bytes of messages free for use by the collection software.

Seecube writes out two records for the *sendw* and *recvw* utilities. One record lists the beginning of the call and the other record pertains to the end of the service. The major differences between the two records are the times and cross-referencing information. For the *send* and *recv* services, it seems that three records can be written [4:28, 32-33]. The first and third records correspond to the beginning and ending of the events respectively. Thus, the only real difference between the services with waits and the services without waits is the extra Seecube record.

In reference to translation to Seecube format, *send* events were mapped to *sendw* events and *recv* events were mapped to *recvw* events. In the case of a *recv*, the begin time was the time just prior to the call to the service and the end time was the time that *status* was called and the returned value showed that all was complete. This then in some sense maps to a *recvw*. In the case of a *send*, the results are different than what it appears Seecube would do. Seecube evidently also requires a *status* to clear a *send*. In the case of PRASE, the begin time is compatible but the end time is the time just after the *send* service completes. A note was placed in the User's Manual pertaining to this. PRASE could be changed to require a user to call *status* until the *send* completes, however, this was not done.

The new format will not keep the Seecube cross-referencing information (specifically, extra bytes are not required to be sent with a message) because it can be derived if needed. However, the count field in the new format is used for cross-referencing. The time information will be kept in two fields, one for the beginning time and one for the ending time.

Seecube's Data Collector yields a file containing records with 14 fields. To aid the discussion, two example records are provided. For more details on the Seecube file format, refer to the appendix in this thesis pertaining to that subject.

Record Pair Example

```
0  4  5  6  3480  wb  139  10  256  18  7  18  17  x
0  4  6  5  3492  we  139  10  256  18  7  18  17  x
```

With this example in mind, we will now give an example record in the new format. One record in the proposed format should correspond to two records in the above example. The proposed format can contain up to 12 fields. For further information about this format, refer to the appendix pertaining to PRASE file formats. An example of this record format follows:

```
sendw   0   4   3480   3492   139   10   256   1   18   7   -1
```

Now we can discuss the translation process from the new format to the Seecube format. In using Seecube, we are mainly interested in displaying information pertaining to message passing. Thus, only channel opens, channel closes, message sends, and message receives will be translated. Wrap records are important for translation but do not get translated into any type of record for the Seecube Resolver to use. All other fields will discarded in reference to translation.

It is now time to discuss how the single new record can translate into the two example records in Seecube format. The translation of several of the fields are very straightforward. For example, the *sendw* maps directly to the *wb* and *we* of the two Seecube records. The 0 (node) in the new record maps to field 1 in both Seecube records. The 4 (*pid*) maps to field 2 (both records). The 3480 (start time) maps to field 5 in the first Seecube record and the 3492 (end time) maps to field 5 in the second Seecube record. The 139 (channel) maps to field 7 in both records. Also the 10 (message type) and 256 (message size) map to fields 8 and 9 respectively for both Seecube records. The 18 (node the message was sent to) and 7 (pid the message was

sent to) map to fields 10 and 11 respectively for both records. Finally, the -1 (text marker) maps to field 14 in the Seecube format of both records.

At this point we have discussed filling in several fields for the Seecube format. The following two records show the fields dealt with thus far and leaves dashes in the fields we still need to fill in.

```
0  4  -  -  3480  wb  139  10  256  18  7  -  -  x
0  4  -  -  3492  we  139  10  256  18  7  -  -  x
```

First, we will discuss fields 3 and 4 of the Seecube format. Field 3 is the sequential numbering of this event within the trace for a specific node. Field 4 is the number that points to the associated record that corresponds to the current record.

Before going further, a point should be clarified. A receive event means that something was read by a Hypercube call or was cleared as being read by a Hypercube call. It does not refer to the actual time that the message arrived at the physical node.

As Seecube records are built, a running count determines what sequential number the current record is. For this to work, the records in the new data format must have been put in the file in the order that they occurred. As long as records are kept in order in memory, and then dumped to file in order, the records for the individual nodes will be in order. Thus, the *wb* record number (field 3) can be derived by counting. Field 4 in *sendw* and *recvw* types of services will show correspondence to a contiguous record since no events should occur between the start and ending of these services. For *send* types, they also should have no occurrences between them. A problem comes in when we try to deal with *recv* records because of the way that they are collected. When a *recv* is issued no record is saved at the time. This is a problem as will be demonstrated shortly. For non-blocking services, Seecube records

a start record and an ending record. The time that the *recv* is issued is saved. Then, when a *status* is called that completes this *recv*, the ending time is included and the record is written to the trace. If any services have been called between the time that the *recv* was initiated and the time that it completed, we then have a time that is out of order. In fact, a service must be called in between - the *status* service is called to complete the *recv*. The User's Manual will reflect the fact that to use Seecube, you must not call any message services between the start and stop of a *recv* so that the timing in the *human* file will not get out of order. If this is adhered to, the results should be fine and our ordering assumption will stand.

Field 4, then, should contain a number that is one larger than that of field 3. In the ending record (*we* for Seecube), the fields can be derived in like manner. Thus, field 3 will contain the next sequential number and field 4 will point back to the previous record.

Fields 12 and 13 are the last two fields needed to make the translation from the new data format to the Seecube data format. These fields correspond to the receive records on the node that the message was sent to. As previously stated, records that pertain to a single node are in sequential order.

Fields 12 and 13 are the most difficult to deal with since messages sent in one order can be received in a different order if their message types are different. If the message types are the same, however, the iPSC will keep the messages sent from one node to another in order all the way to the receiving node. This fact makes filling in these fields possible. If the iPSC ever switches schemes, these fields will no longer be translatable. A count is included in the new format that is a count specific to the corresponding node, pid, and message type. The count is incremented by one from the previous value every time the triplet of information matches. Thus, for each specific triplet, we keep a separate count.

To cut through some of the possible confusion here, we will illustrate the concept with an example. Node 1 has 5 messages to send to node 7. They are labeled

as A, B, C, L, and M. Node 1 also has 3 messages for node 12. These are labeled X, Y, and Z. After the run, we wish to translate our recorded data into Seecube format so we can analyze the data using the Seecube Sequencer. A, B, and C are all of type 100. L and M are of type 200; and X, Y, and Z are also of type 100.

Let's say now, that node 1 sends messages A, B and C to node 7. Almost immediately after C is sent, node 1 fires off messages L and M also destined for node 7. Soon thereafter, messages X, Y, and Z are also dispatched to their destination.

The send trace is well defined and could be recorded in only one way because events are recorded in order on each node. Thus, the trace should look like the following (with times, sizes, etc. dummied).

rec Type	nod	pid	start time	end time	chan	msg type	msg size	seq cnt	other nod	other pid	mrk
sendw	1	0	543	548	170	100	1024	1	7	0	-1
sendw	1	0	548	548	170	100	1024	2	7	0	-1
sendw	1	0	553	558	170	100	1024	3	7	0	-1
sendw	1	0	558	563	170	200	445	1	7	0	-1
sendw	1	0	563	563	170	200	768	2	7	0	-1
sendw	1	0	598	603	170	100	1024	1	12	0	-1
sendw	1	0	603	603	170	100	1024	2	12	0	-1
sendw	1	0	608	613	170	100	1024	3	12	0	-1

Notice that the sequence counts increase for each message that has a matching triple of message type, other node, and other pid. With the above records in mind, we can now talk about the receive information that might be recorded. Node 7 would have to record receive information for message C after message B since message C was sent after message B and is of the same message type. Message L however, could be recorded before or after a receive for either message B or C. This is because the Hypercube will only keep messages with like types in order from one specific node to another specific node. Thus, message L could actually be received prior to messages A, B, or C. Let's propose a set of receive records for sake of our example.

rec Type	nod	pid	start time	end time	chan	msg type	msg size	seq cnt	other nod	other pid	mrk
recvw	7	0	585	590	320	100	1024	1	1	0	-1
recvw	7	0	590	590	320	200	445	1	1	0	-1
recvw	7	0	590	595	320	200	768	2	1	0	-1
recvw	7	0	595	605	320	100	1024	2	1	0	-1
recvw	7	0	610	615	320	100	1024	3	1	0	-1

Thus, we see that even though messages were received out of the order they were sent, like message types are received in order. To correlate the send and receive records, then, we simply match the send and receive nodes, the message count and the sequence count. With these six fields matched, we are able to correlate send records on one node with receive records on another node. The only problem would be if the counts on one node wrapped around while the counts on another node did not. Wrap records are written to the trace when a wrap occurs so that the translator can determine if the counts are actually the ones that go together or not.

If we always began recording events at the start of a run, we wouldn't really need the sequence count field. We could figure the count out as we were translating. We need this count, however, so that we can start recording at times other than start time. The count will be incremented even when an event is not recorded thus keeping the needed sequence information.

Since messages are kept in order within message type, and we are keeping a count for each message type to each node-pid combination, we now have a way of producing the same sequential number on both a send and receiving node for matching triples of information. This means that each node then figures out the numbers that allow us to correlate send and receive records of information.

Once the correlation is made, fields 12 and 13 of the *send* get the values of fields 3 and 4 of the *receive* ending record. Fields 12 and 13 of the *receive* will get

the values of fields 3 and 4 of the *send* beginning time and the translation will be complete.

In an attempt to be a little more formal, I will introduce some mathematical concepts and symbols. First, let us define events occurring some node and process as

$$E_{n_h,p_i,j} \tag{G.1}$$

where n_h is the node h and p_i is the process i the event occurred on and j is the sequential number of the event. $T0$ denotes startup time of the parallel job and

$$T(E_{n_h,p_i,j}) \tag{G.2}$$

denotes the time that event j occurred on node n_h and process p_i. All times are absolute times. Since records are written sequentially into memory (and then to file) in the order that they occur, the following holds true

$$T0 < T(E_{n_h,p_i,j-1}) < T(E_{n_h,p_i,j}) < T(E_{n_h,p_i,j+1}) \tag{G.3}$$

Thus, events are stored in a known order that can later be used to our advantage. Now on to matching *send* events on one process with *receive* events on another process. Since two processes p_l and p_m running on different processors would both start at or sometime soon after T0, all events must occur after T0. We also know that events are stored in order of occurrence (from Equation 1). This means that the time of the first *send* event s from p_l to p_m is stored prior to the next *send* event time $s + k$ from p_l to p_m. Mathematically this is stated as

$$T(E_{n_a,p_l,s}) < T(E_{n_a,p_l,s+k}) \tag{G.4}$$

We also know that the first receive event time r on p_m is stored in the trace prior to the second receive event time $r + c$, or

$$T(E_{n_b,p_m,r}) < T(E_{n_b,p_m,r+c}) \tag{G.5}$$

The question is, "did the first message sent m_1 arrive prior to the second message sent m_2?" The iPSC Hypercube works to our advantage in answering this question. If the two messages sent have different message types, then the order of receipt is unknown prior to actual receipt. Either message could get to its destination first. This is not a problem, however, since the message types would differ in the trace records and matching could occur in only one way anyway. If, however, the message types are the same, then m_1 will always proceed m_2 in arriving. Thus, for any message traces that could cause an exact match (meaning message types are the same), all the traces will be in the proper order.

Due to the iPSC ordering within message type, then, we can say that the first send from one node and process correlates to the first matched receive on some other process, not necessarily on another node. Thus, these orderings and user restrictions allow us to rebuild fully the Seecube records required for compatibility.

Appendix H. *Configuration File Example*

This appendix is being used as a pointer. To examine a copy of the configuration file, refer to the User's Manual Appendix A.

Appendix I. *Summary Paper*

On the next pages you will find a summary paper for this project. Appendix numbering is not used.

PRASE: Intel iPSC Hypercube Instrumentation Software

Thomas C. Hartrum
Mark A. Kahl

December 1, 1988

1 Introduction

Single processor computers are rapidly becoming a thing of the past. Their replacements are multiple processor systems, running in parallel, which may yield greater overall speed and efficiency. Mainframe computers sometimes use dual CPUs and almost always contain separate specialized processors to interface to input/output devices such as disk drives. Even some smaller home computers employ specialized chips (processors) to handle functions such as input/output or graphics.

There is great potential, in a parallel environment, to complete several times the amount of work of a single processor in the same amount of time. Just as a factory might hire more workers to increase production, computers can utilize multiple processors to better handle their workloads. In fact, when chip makers reach speed limitations, the only alternative for faster service may be to do more work in parallel.

The potential speedup offered by a parallel environment can be appealing to a user. A programmer, however, may not share the user's enthusiasm when faced with events happening at the same time and often in an unpredictable fashion. If parallel processors worked autonomously, the problem might not be so complex. Normally, though, they interact with each other, causing much greater complexity.

One way to attack the potential confusion is to use a parallel processing monitor/analyzer. **PRASE** is a suite of software that provides such a

monitoring capability for the Intel iPSC Hypercube. The goal of the project was to support the user community in their parallel processing efforts with a working monitor that people would be able to use easily.

The software from an existing project at Tufts University in Massachusetts, Seecube [1], was obtained as a starting point for this project. Seecube has three major parts [1, 1]. The first is the *Data Collector*, a set of subroutines that replace the normal Hypercube calls with calls to instrumentation routines that take care of data collection. Once the data is collected, the *Resolver* "cross-references these traces by matching sends with their corresponding receives, and sorts the traces into a single global trace for the entire hypercube" [1, 1]. The *Sequencer* is the graphical display portion of Seecube [1, 1]. It provides several different displays to aid the parallel processing user in understanding the workings of his code.

Limitations of Seecube included the unavailability of support for the current version of the Intel iPSC; lack of ability to collect application-specific data; the requirement for modification of user data structures (specifically messages); the requirement for manual instrumentation of the user's code; and the need for some additional graphics routines. However, Seecube provides many excellent graphics displays that are very useful. Therefore, compatibility with Seecube was desired in order to take advantage of Seecube's graphical capabilities.

A new data collector was written, patterned somewhat after Seecube's. The **PRASE** collection routines are written in C and can be called from both C and Fortran application programs. Like Seecube, the normal Hypercube routines must be replaced by calls to the **PRASE** routines. To aid the user with the task of code instrumentation, preprocessors for both C and Fortran were written. These programs relieve the user of most of the work required to prepare source code for instrumentation. Ada was used as the implementation language for the preprocessors.

Since **PRASE** data formats are different from Seecube data formats, a translator was required to maintain the desired compatibility. The translator produces a file that is usable by the Seecube *Resolver*. Figure 1 depicts the **PRASE** system and its compatibility with Seecube.

[1] At this point in the **PRASE** implementation, certain cases require the user's code to be changed or added to.

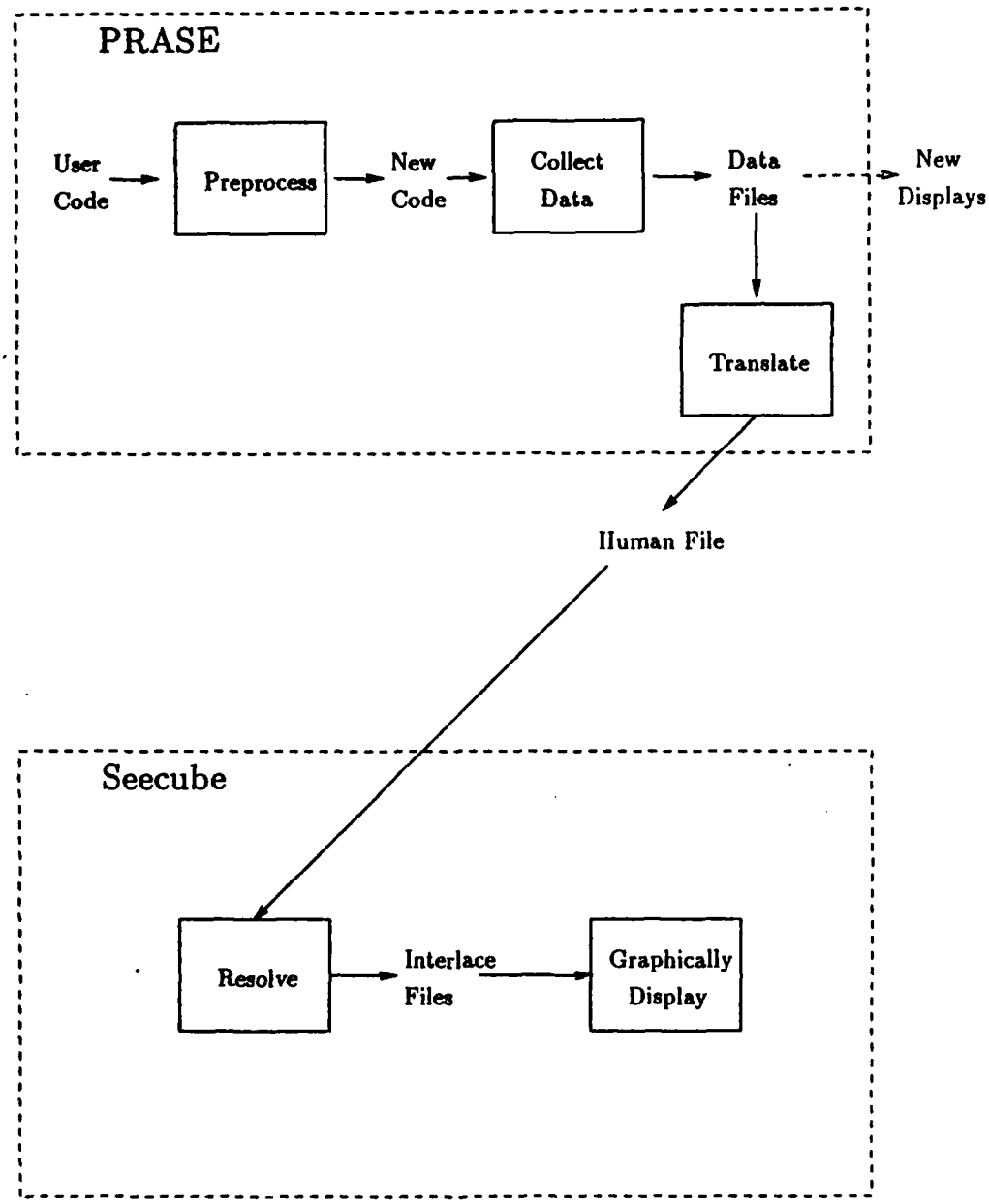


Figure 1: **PRASE** with Compatibility to Seecube[1]

2 Data Collection

A set of subroutines comprise the data collection mechanism utilized by **PRASE**. These routines call the desired Hypercube services while insuring that performance data is collected. One set of subroutines, written in C, support both C and Fortran programs. This means that the Fortran user will ultimately access the service provided by the C node library as opposed to the Fortran node library.

Both single and multiple processes per node are supported. However, only messages sent to specific nodes are supported; global sends to the entire cube or subcubes are not supported. A user can gather data on the *cclose, copen, flick, greenled, probe, recv, recvw, redled, send, sendw, status,* and *syslog* Hypercube calls.

The collection software attempts to synchronize node clocks so that some sense of global timing can be achieved. Note that Seecube requires some type of global time stamp although their synchronization method is different [1, 2]. A single process is designated as the Controlling Process (CP) which takes command of the synchronization step. CP is a process running on the node with the lowest node number. CP notifies a single process on each of the other nodes that it is their turn to synchronize. As a node is polled by CP, synchronization can begin. The polled node saves the current time and then sends a message to CP. CP in turn puts its current time and its base time (start time) into the message and sends it back to the node. The node saves the time of receipt and then calculates an approximate difference between its clock and CP's clock. This difference, along with CP's base time, can then be used to determine an approximate global time for each data collection event. The local process then passes the time information on to any other processes running on the same physical node. At this point, synchronization is complete for a single node.

Besides Hypercube system calls, data can also be collected on user specific information. Routines have been included that allow a user to save values of variables specific to his program. Several variable types are supported including integer, long integer, float, double, and character. This capability gives the user flexibility in what he can monitor. If the change of a variable over time is significant, a user can gather data pertaining to this change.

In reference to message passing, the available version of the Seecube Data Collector required a user to leave the first 4 bytes of each message available for "cross-referencing information" [2, 4]. In order to increase user transparency, PRASE uses another means to develop the cross-referencing data required by the Seecube display software.

There are six pieces of information that uniquely identify a message from any other message for a single run. They are the node and pid from which the message was sent, the node and pid to which the message was sent, the message type, and a unique sequence number. The sequence number is a count of other messages with the first five pieces of identical information. For example, the first message of type 10 sent from node 1 pid 5 to node 20 pid 3 would be assigned a sequence count of one. A second message of the same type, origination point, and destination would get a count of two. Each node keeps track of these sequence counts and logs the information into the **PRASE** records as sends and receives complete. Since the Hypercube keeps messages of the same type *from* a specific location *to* a specific location in order, the order in which the messages are sent corresponds to the order in which they are received. Therefore, as long as both the sending and receiving processes are keeping track of these sequence counts, the cross-referencing information can be determined after the run is complete.

To actually get the data from the nodes to files on the host, a special host process runs in the background that accepts data collected on the nodes (see Figure 2). Since this process runs until killed, data can be dumped to it at any time during a run. **PRASE** can be set up tc save all data in memory until a process is ready to terminate; dump data after every record that is collected; or dump data periodically throughout a run.

This does several things for a user. First, if time is an issue, data can be saved in memory. If time is not that critical, great amounts of data can be collected by dumping periodically. This also means that the user does not have to give up a lot of space on a node for gathered data. Instead, the data can be dumped often so that space is taken up on disk as opposed to in main memory. If a process is terminating abnormally, **PRASE** can be set to dump after every record to aid in debugging.

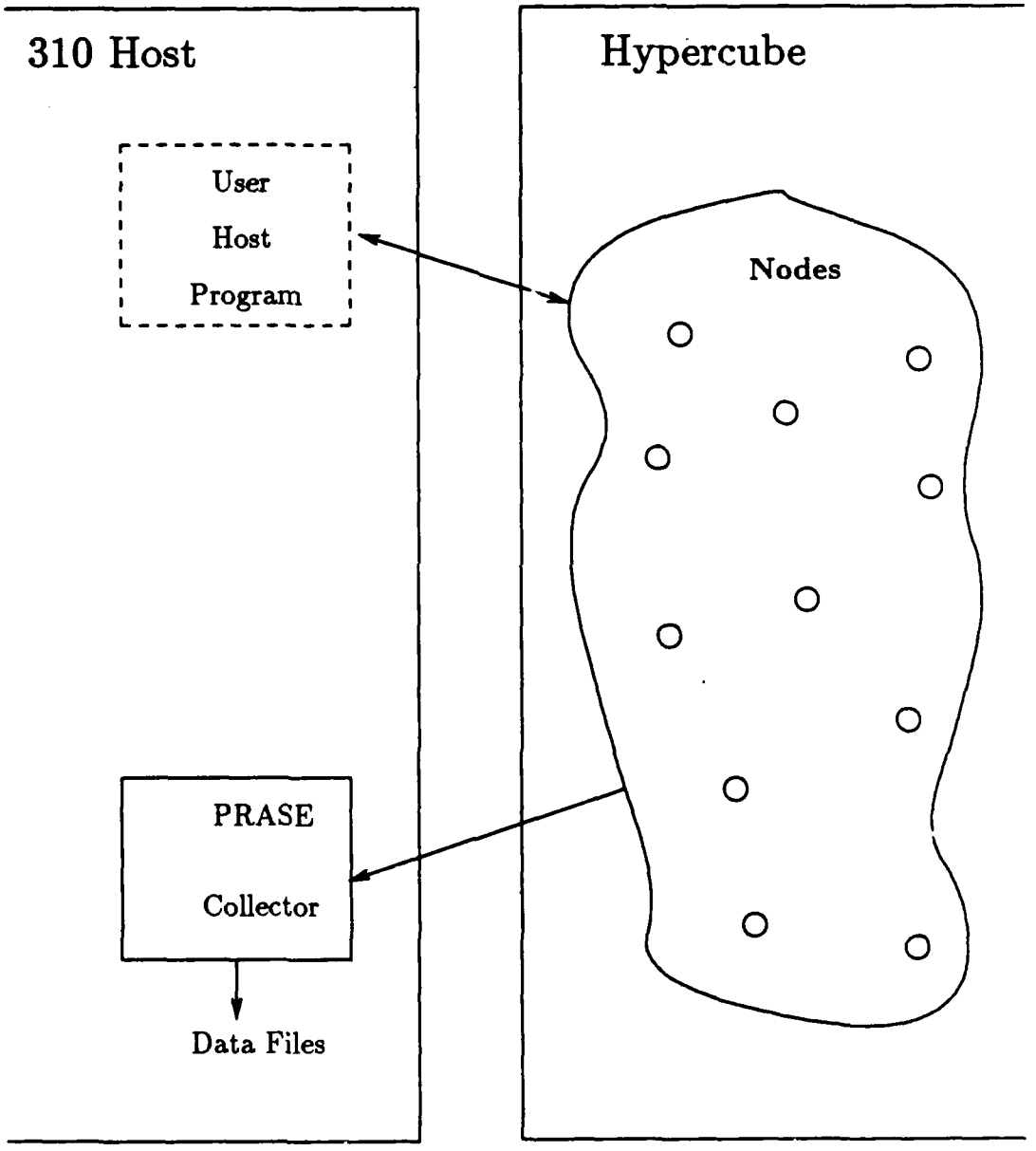


Figure 2: Data Collection using a Background Process

3 Preprocessing

With ease of use as a goal, it was decided early that preprocessors were important. The preprocessors require no interactive input. Instead, the user communicates with the programs through a configuration file that contains the needed information. The same configuration file used in the preprocessing step is also used for translation.

The configuration file normally provides five pieces of information to the system. Information is provided to tell the system which nodes will be used in a particular application. Process id's (pids) are also given in this file. A collection start time can be provided to **PRASE**. This is a relative time (from the beginning of a run) that determines when data collection will begin. The configuration file also includes a list of the Hypercube calls for which the user would like to gather data. For example, if a user wants to examine send and receive message services, the appropriate information must be included in this file. Finally, the names of all code files to preprocess must be included.

4 Seecube Compatibility

A translator has been implemented to take the **PRASE** data and convert it to a Seecube format. This converted data can then be used by the Seecube system so that its graphical display capabilities can be utilized.

Data collected pertaining to items other than *send, sendw, recv, recvw, cclose,* and *copen* is discarded. The translator is responsible for determining the message cross-referencing information required by Seecube. It will correlate *send* records to *receive* records.

It should be noted that **PRASE** handles the *send without wait* service differently than Seecube. **PRASE** collects a start time for *send* and, once the call to the actual send routine completes, records an end time. The send service, however, does not guarantee that the message buffer is free when it returns control to the caller. The **PRASE** end time reflects the end time of the call, not the time a buffer was freed. Seecube assumes this end time to be the time a buffer was free. Therefore, as a user reviews the Seecube displays, he should interpret them in light of the way **PRASE** has collected the data.

5 Summary

We have provided the user a set of tools that allow him to instrument his code. Implementing preprocessors, has been the first step in making the system easy to use. Maintaining compatibility with Seecube has provided the graphical capability currently lacking in **PRASE**. Hopefully, with this group of software, a user will be able to gain some insight into the workings of his code. The ability for the user to add application-dependent instrumentation allows for a flexible instrumentation system.

6 Possible Future Work

The acronym **PRASE** stands for Parallel Resource Analysis Software Environment. Although not yet a true software environment, we would like to see it move in that direction. A user interface tool that guides a **PRASE** 'session' from start to finish is envisioned.

Several enhancements are needed to the current software. At this time, the Fortran preprocessor does not totally support Fortran. The C preprocessor requires the user to add a special comment. Both preprocessors shall be extended to provide total transparency to the user. The data collection routines and translator also require enhancements to ease current restrictions placed on the user. In addition, we would like to explore other graphical display techniques, specifically three dimensional displays that make use of depth queues in displaying results.

7 Acknowledgements

We would like to thank Alva Couch and Tufts University for their willing support.

References

[1] Couch, Alva L. and others. *An Interactive System for Analysis of Hypercube Message Passing Performance.* Technical Report, Department of Computer Science, Tufts University, Medford, MA, 1986.

[2] Couch, Alva L. *Seecube User's Manual.* Department of Computer Science, Tufts University, Medford, MA, 24 November 1987.

Appendix J. *Volume II Pointer for Code*

The code has not been incorporated into this document. It has been placed in Volume II. This appendix is being used as a pointer to where the code may be found. For information pertaining to the code associated with this thesis, contact:

Dr. Thomas C. Hartrum
Dept. of Electrical & Computer Engineering
Air Force Institute of Technology
AFIT/ENG
Wright-Patterson AFB, OH 45433-6583

Bibliography

1. Bailor, Capt Paul. Student. Personal Interview. Air Force Institute of Technology. 1988.
2. Couch, Alva L. and others. *An Interactive System for Analysis of Hypercube Message Passing Performance.* Technical Report, Department of Computer Science, Tufts University, Medford, MA, 1986.
3. Couch, Alva L. *Seecube User's Manual.* Department of Computer Science, Tufts University, Medford, MA, 24 November 1987.
4. Couch, Alva L. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* Technical Report 88-4. Department of Computer Science, Tufts University, Medford, MA, April 1988.
5. Donlan, Capt Brian. Instructor. Personal Interview. Air Force Institute of Technology. 1988.
6. Ferrari, Domenico and others. *Measurement and Tuning of Computer Systems.* Englewood Cliffs NJ: Prentice-Hall, Inc., 1983.
7. Hearn, Donald and M. Pauline Baker. *Computer Graphics.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1986.
8. Hodges, Capt Bill. Instructor. Personal Interview. Air Force Institute of Technology. 1988.
9. Kerola, Teemu and Herb Schwetman. *Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs. Performance Evaluation Review - Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems 15:* No. 1 (Special Issue) 163-174. (May 1987) (Conference was held at Banff, Alberta, Canada, May 11-14, 1987)
10. Meyers, Glenford J. *The Art of Software Testing.* New York: John Wiley & Sons, 1979.
11. Norris, Richard. Personal Interview. Systems Analyst. System Research Lab, Dayton OH, 1988.
12. Rowe, Capt Janice F. *A Network Monitoring Facility for a Distributed Data Base Management System.* MS thesis, AFIT/GCS/EE/85D-14. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1985.
13. Tufte, Edward R. *The Visual Display of Quantitative Information.* Cheshire Connecticut: Graphics Press, 1983. (Fourth printing in January 1985)

Vita

Captain Mark A. Kahl [redacted] attended Palomar Junior College until entering the USAF in April 1976. He served as a Communications Center Specialist until he began attending California State University at Sacramento through the Airmans Education and Commissioning Program in 1981. He received the degree of Bachelor of Science in Computer Science in December 1983. He subsequently attended Officer Training School, graduating in April of 1984. He served as a Computer Analyst Communications Software Engineer at the Cheyenne Mountain Complex Colorado from May 1984 until entering the School of Engineering, Air Force Institute of Technology in May 1987.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No [illegible]-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/88D-11	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, WPAFB, OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: STRATEGIC DEFENSE INITIATIVE ORG.; 8b. OFFICE SYMBOL (If applicable): S/PI	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): The Pentagon, Washington DC 20301-7100	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

PRASE: INSTRUMENTATION SOFTWARE FOR THE INTEL IPSC HYPERCUBE

12. PERSONAL AUTHOR(S)

Mark Albert Kahl, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1988 December	210

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Parallel Processing, Monitors, Instrumentation
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: Dr. Thomas C. Hartrum
Associate Professor of Electrical and Computer Engineering

[signature] 12 Jan 1989

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Thomas C. Hartrum, Associate Professor	22b. TELEPHONE (Include Area Code): (513) 255-2024 / 22c. OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

ABSTRACT:

PRASE (Parallel Resource Analysis Software Environment), developed at the Air Force Institute of Technology to support local users, consists of a set of subroutines and programs that aid a user in monitoring parallel processing software targeted for an Intel iPSC Hypercube. PRASE was in many ways patterned after Seecube, an effort by Alva Couch and others at Tufts University in Massachusetts. Like Seecube, instrumentation code must be embedded in a user's source code to facilitate data collection. After data has been collected, a translator may be used to translate PRASE data into Seecube format. Once translated, existing Seecube software can be utilized to produce several kinds of graphical displays on a Sun workstation.

Seecube, however, is not required to be able to use PRASE. PRASE allows a user to gather data on several processes per node and gives the user the capability to collect information on variables specific to his program. This allows application-specific instrumentation. Preprocessors for both C and Fortran automatically embed necessary subroutine calls using a user defined configuration file. The data collection subroutines are written in C and can be called by both C and Fortran. Data collected during program execution can be held in Hypercube memory and written to disk at the end of a run, or dumped periodically to disk during a run which may aid in debugging. The resulting data files can then be translated to Seecube format or used as input to other data analysis and display programs. The two preprocessors, as well as the translator were implemented in Ada.